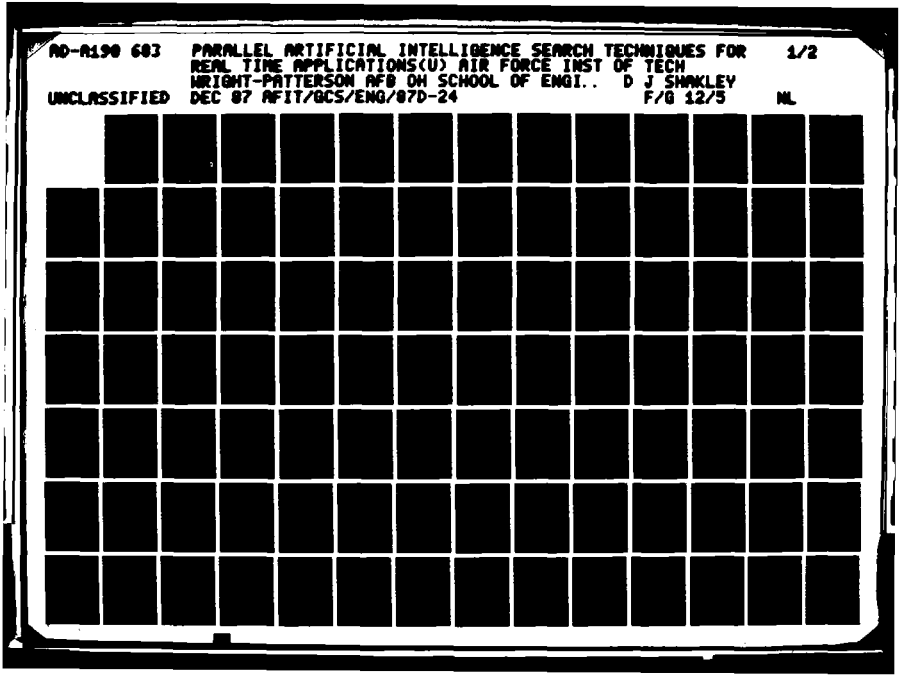
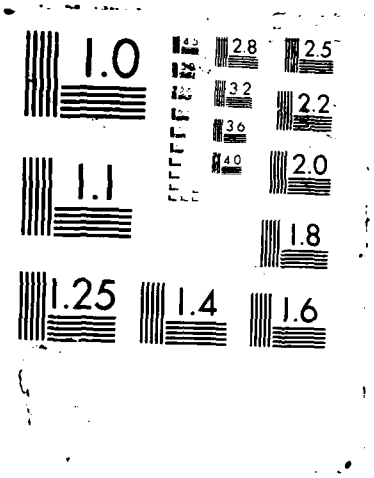

AD-A190 683 PARALLEL ARTIFICIAL INTELLIGENCE SEARCH TECHNIQUES FOR REAL TIME APPLICATIONS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. D J SHAKLEY
1/2
UNCLASSIFIED DEC 87 AFIT/GCS/ENG/87D-24 F/G 12/5 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

PARALLEL ARTIFICIAL INTELLIGENCE SEARCH TECHNIQUES FOR REAL TIME APPLICATIONS

THESIS

Donald J. Shakley
Captain, USAF

AFIT/GCS/ENG/87D-24

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/87D-24

PARALLEL ARTIFICIAL INTELLIGENCE SEARCH TECHNIQUES FOR REAL TIME APPLICATIONS

THESIS

Donald J. Shakley
Captain, USAF

AFIT/GCS/ENG/87D-24

AFIT/GCS/ENG/87D-24

PARALLEL ARTIFICIAL INTELLIGENCE SEARCH TECHNIQUES FOR REAL TIME APPLICATIONS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

Donald J. Shakley, B.S.

Captain, USAF

December 1987

Preface

This study was prompted by the need for expert systems to perform in real-time for systems like the Robotic Air Vehicle. Speedups are necessary if expert systems in the form of production systems are to be used. This study looks at simple production system control rather than at the complex Rete algorithm. It is hoped that through simplicity, speed can be gained and can be predicted. The predictability is important in real-time were additional processing power could be brought to bear to increase the response time.

This work would not have been possible without the assistance, advice and support of many people. In particular, I would like to thank my thesis advisor, Dr. Gary Lamont, without whom's help this effort would have been impossible. Also my committee members, Major Steve Cross and Captain Nate Davis, have provided guidance and influence at key stages of my effort. None of this would have been possible without the support and understanding of my family and friends.

Donald J. Shakley

Table of Contents

List of Figures

Abstract

State space search is an important component of many problem solving methodologies. The computational models within Artificial Intelligence depend heavily upon state space searches. Production systems are one such computational model. Production systems are being explored for real-time environments where timing is of a critical nature. Parallel processing of these systems and in particular concurrent state space searching seems to provide a promising method to increase the performance (effective and efficient) of production systems in the real-time environment.

Production systems in the form of expert systems, for example, are being used to govern the intelligent control of the Robotic Air Vehicle (RAV) which is currently a research project at the Air Force Wright Aeronautical Laboratories. Due to the nature of the RAV system, the associated expert system needs to perform in a demanding real-time environment. The use of a parallel processing capability to support the associated computational requirement may be critical in this application. Thus, parallel search algorithms for real-time expert systems are designed, analyzed and synthesized on the Texas Instruments (TI) Explorer and Intel Hypercube. Examined is the process involved with transporting the RAV expert systems from the TI Explorer, where they are implemented in the Automated Reasoning Tool (ART), to the iPSC Hypercube, where the system is synthesized using Concurrent Common LISP (CCLISP). The performance characteristics of the parallel implementation of these expert systems on the iPSC Hypercube are compared to the TI Explorer implementation.

The implementation on the iPSC hypercube points out the feasibility of implementing a production system in CCLISP and gaining performance improvements over the TI Explorer. This study shows poor performance speedups due to poor load balancing combined with a large communication overhead in contrast to the problem size.

Parallel Artificial Intelligence Search Techniques For Real-Time Applications

I. Introduction

Real-time applications exist that involve "hard" problems that currently defy generic algorithmic approaches. Thus, problem solving paradigms from Artificial Intelligence (AI) are being applied to these applications. Due to the computational complexity, however, these approaches have poor computer performance characteristics (Gupta, 1986). Parallel processing seems to offer a possibility to improve computational performance for hard real-time problems. The purpose of this first chapter is to provide a background for the major components of this study: Artificial Intelligence, Parallel Processing, and Real-Time Processing. This chapter also defines and scopes the problem and the approach used in this study.

Artificial Intelligence

Artificial Intelligence (AI) is concerned with the designing of computer systems that exhibit intelligent characteristics of human behavior. These methods are used when other direct approaches start to deteriorate due to a lack of generality of solution. Examples of such behavior include language understanding, reasoning, and problem solving (Barr and others, 1981). These problems are studied in AI by using a computational model. Many computational models exist for AI problems. A computational model is a formalism used to describe a method of solution. These models present different ways to represent the problem domain. Examples of these models include production systems, semantic networks, frames, and logic (Fischler and Firschein, 1987).

For each of these representations the method of solution can be formulated as a state space search. A state space should be defined for the problem domain as an essential paradigm component. State-space definition also requires a set of initial states, a set of goal states and a set of rules for getting from one state to the next state. The state space can then be thought of as a graph with nodes (vertices) corresponding to states. An example of a state space can be seen in Figure 1-1. The explicit graphical representation

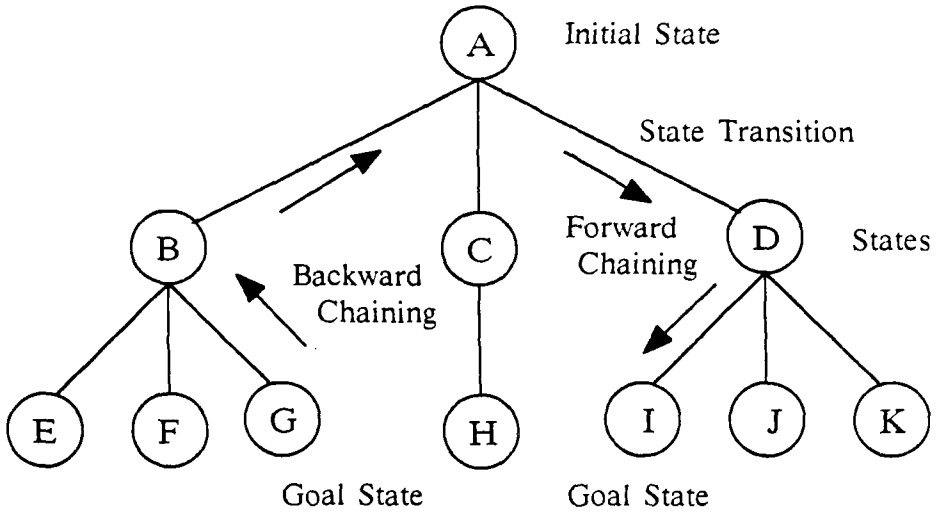


Figure 1-1. State Space Representation.

of a state depends upon an individual selection of an associated data (node) structure. The problem solution can then be determined by searching the graph either from an initial state to a goal state (forward chaining) or from a goal state to an initial state (backward chaining).

The production system fits the definition of a state space search in the most generic terms. A production system consists of an initial set of facts and a set of rules that

operate on the facts. Only the goal states are not explicitly defined. The production system stops when no rules can be applied. This is the basic formulation of a production system with its roots dating back to Post in 1943 (Rich, 1983). A production system has the ability to represent any Turing computable function (Post, 1943). This class of functions include all the primitive recursive functions as well as partial recursive functions (Manna, 1974). Most, if not all, practical functions are primitive recursive functions. Since any of the other representations used in AI are primitive recursive functions they can then be formulated as a production system.

Production systems are most prevalent in AI as expert systems. These systems can be used in different application areas. Application areas for production systems include prediction, diagnosis, design, planning, monitoring, debugging, repair, instruction, and control (Waterman, 1986).

Parallel Processing

Parallel processing is the use of more than one processing element to compute the solution to a problem. By using more processing elements, it is hoped that the time to solve the problem will be reduced over the time to solve the problem on a single processor. This is done at the expense of space efficiency. However, sometimes a parallel architecture could add space that would not normally accessible on a serial architecture.

There are several ways to achieve performance improvements in computer systems besides parallel architectures: faster hardware technology, improved serial architectures, better algorithms and code optimization. There are several reasons for looking toward parallel architectures. First, parallel architectures can evolve as fast as hardware technologies become available. Second, many problems associated with AI are computationally "hard" (exponential time-order) or NP-complete (see Appendix C). If a problem is NP-complete, this implies that time-order improvements in solution algorithms are unlikely due to many years of computational studies (Aho and others, 1974). It should be

noted that parallelism can not produce polynomial time solutions to exponential time problems (Norman, 1985). But, it is possible to improve the constant term of the time complexity. Also, the exponential time bound is often times worst case. In AI problems, the use of heuristics can reduce the time complexity of state space searches. Third, some problems seem to lend themselves to parallel solutions because the problems decompose easily into independent, computationally equivalent pieces. Production systems seem to fall into this category because of the large number of rules that must be matched during each production cycle. The state-space graphical formulation of a production system appears well-suited for parallelism (Gupta, 1986).

A complete examination of parallel architectures is contained in Appendix B. Briefly, however, it should be noted how the major categories of parallel architectures map into the major types of parallel decomposition. The two types of practical architectures according to Flynn's taxonomy is Single Instruction Multiple Data streams (SIMD) and Multiple Instruction Multiple Data streams (MIMD). The two ways to decompose a problem is by function (operations) and by data (objects) (Figure 1-2) (Jamieson and others, 1987). Another method related to data decomposition is object-oriented design (Booch, 1987). This is a method where data entities are viewed as objects. Messages are used to communicate between the various objects. In some cases, a problem can be decomposed in layers of these two methods. In other words, a problem can first be decomposed into partitions by the data, and then these partitions can be decomposed by function (operations) (Figure 1-3). The mapping of decomposition methods to architectures is obvious. The SIMD architectures are designed for lock-stepped operations on multiple data paths that correspond to data decomposition. The MIMD architectures are designed for potentially different operations on multiple data paths that correspond to functional decomposition. The flexibility of the MIMD architectures is useful for the layered decomposition approaches because it allows any operation to be performed on any processor

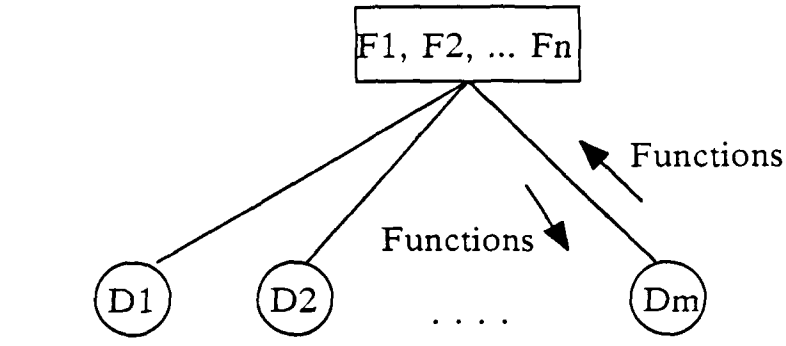


Figure 1-2a. Data Decomposition.

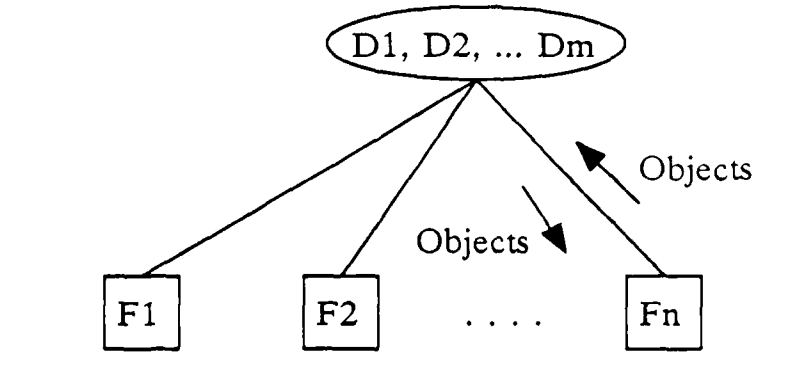


Figure 1-2b. Functional Decomposition.

and with the proper synchronization can simulate a SIMD architecture. This allows nodes within a MIMD architecture to switch between the MIMD mode and SIMD mode.

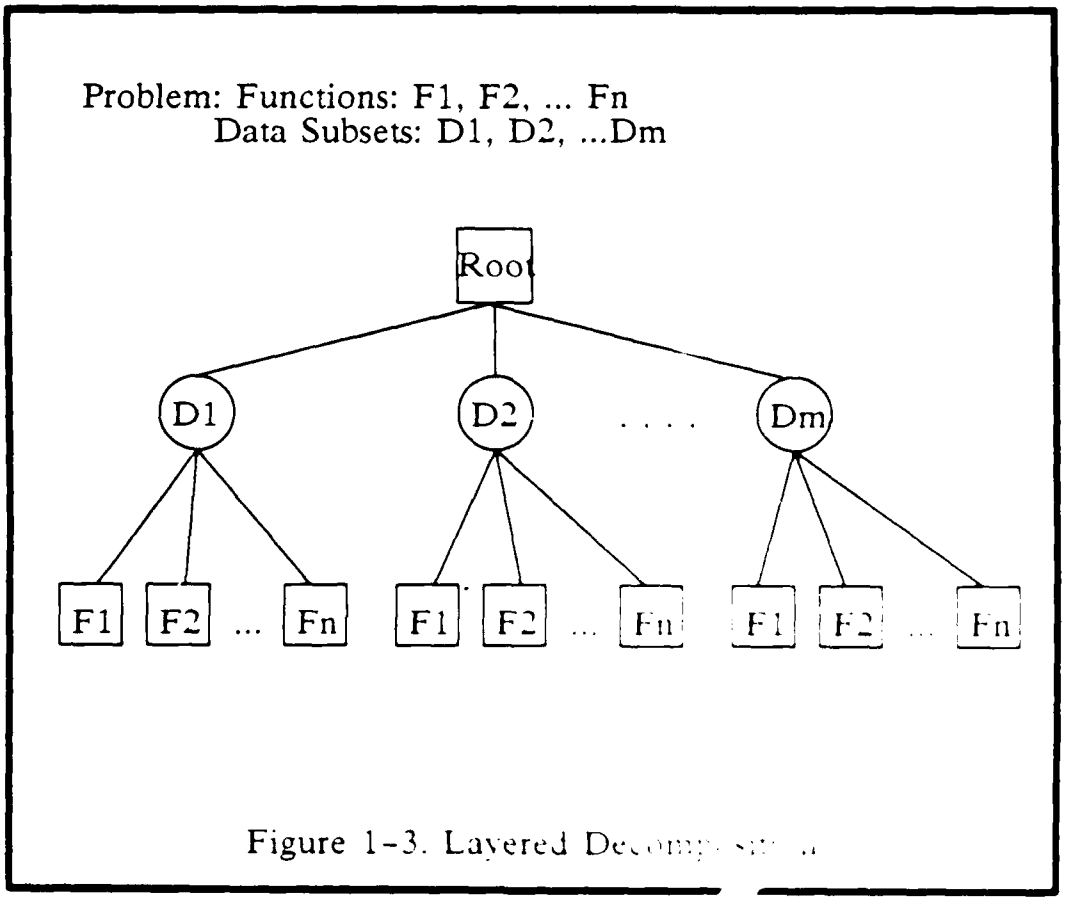


Figure 1-3. Layered Decomposition

Real-Time Processing

There are several important issues in the analysis and design of real-time computer applications. One of the important characteristics is the critical nature of the system execution speed in reference to external events. This can be viewed in terms of the response time of the system to a particular input. For a real-time system, "the time needed to make a calculation has to be less than the time from when the need for the calculation is recognized until the time when the response is needed to take action" (Norman, 1985).

This can vary with the system, but the time is generally relatively small. Relatively small is definitely less than a second and often in the milliseconds or less (Ward and Mellor, 1985).

Another critical characteristic is limited memory capacity. Real-time software typically needs to run in an environment where the size of the program can become a problem. A third consideration is the correctness, reliability and integrity of real-time software. The system needs to run correctly and without failure a high percentage of the time (Ward and Mellor, 1985). These represent the most critical issues dealing with real-time systems.

The problem with a real-time system on a serial architecture is that the execution time and space requirements are relatively fixed for a given operation. A desirable feature of a real-time system would be a variable time and space performance based on the need. With parallel architectures this could be possible. If a problem needed a faster solution based on the time requirement, then more processors could be added to produce the appropriate speedup. This could only be done if the speedup were predictable.

The need for production systems within real-time systems is growing. With parallel processing of production systems, the execution speed is increasing. For real-time systems this speedup needs to be predictable, so that at any given moment more processors can be brought to bear on a problem to decrease the coefficient of the time complexity of the solution.

An example of a real-time application, which is a current research project at the Air Force Wright Aeronautical Laboratories, is the Robotic Air Vehicle (RAV). It is an air vehicle with the capability of autonomous flight operation. This vehicle needs the capability for the "intelligent" control of an air vehicle, the capability to plan and replan missions, and the capability to access flight data on various geographical locations (air-bases, airports, cities, etc). By "intelligent" it is meant that the system can react to conditions rather than fly on a rigid preprogrammed flight path. A diagram of the system

can be seen in Figure 1-4. A complete discussion of the system components can be found in Appendix A.

This study focuses on the "intelligent" flight control components of the system. This component was selected due to its reliance on production systems and its maturity in relation to the entire research project (Graham, 1987). The control of the vehicle can be thought of as a search through a finite state-space over a time period of the vehicle's operation. The problem of intelligent control of a robot is a control-type NP-complete problem (Appendix C) that is best suited to be solved by a production system in real-time (Appendix A). Therefore, this system makes an excellent tool for the study of parallel AI search techniques for real-time applications.

Problem

The RAV system is an example of an intelligent real-time robotic control system implemented using an expert or production system (McNulty, 1987). The purpose of this investigation is to try to increase the performance of the expert system by reanalysis, redesign and reimplementation of the system on appropriate parallel architectures. The hypothesis of this study is that the performance of the RAV expert system can be improved in a predictable and linear manner.

Scope

There are many types of expert systems (Nilsson, 1980). This investigation concentrates on an expert system for intelligent real-time control (Albus, 1981). There is the potential for many types of parallelism within production systems (Douglass, 1985). The search parallelism within a production system is the focus of this effort.

There are many types of parallel architectures (Appendix B). The two type of parallel architectures (MIMD and SIMD) were considered for this study. Only two particular architectures are considered in this study due to availability. A further discussion of

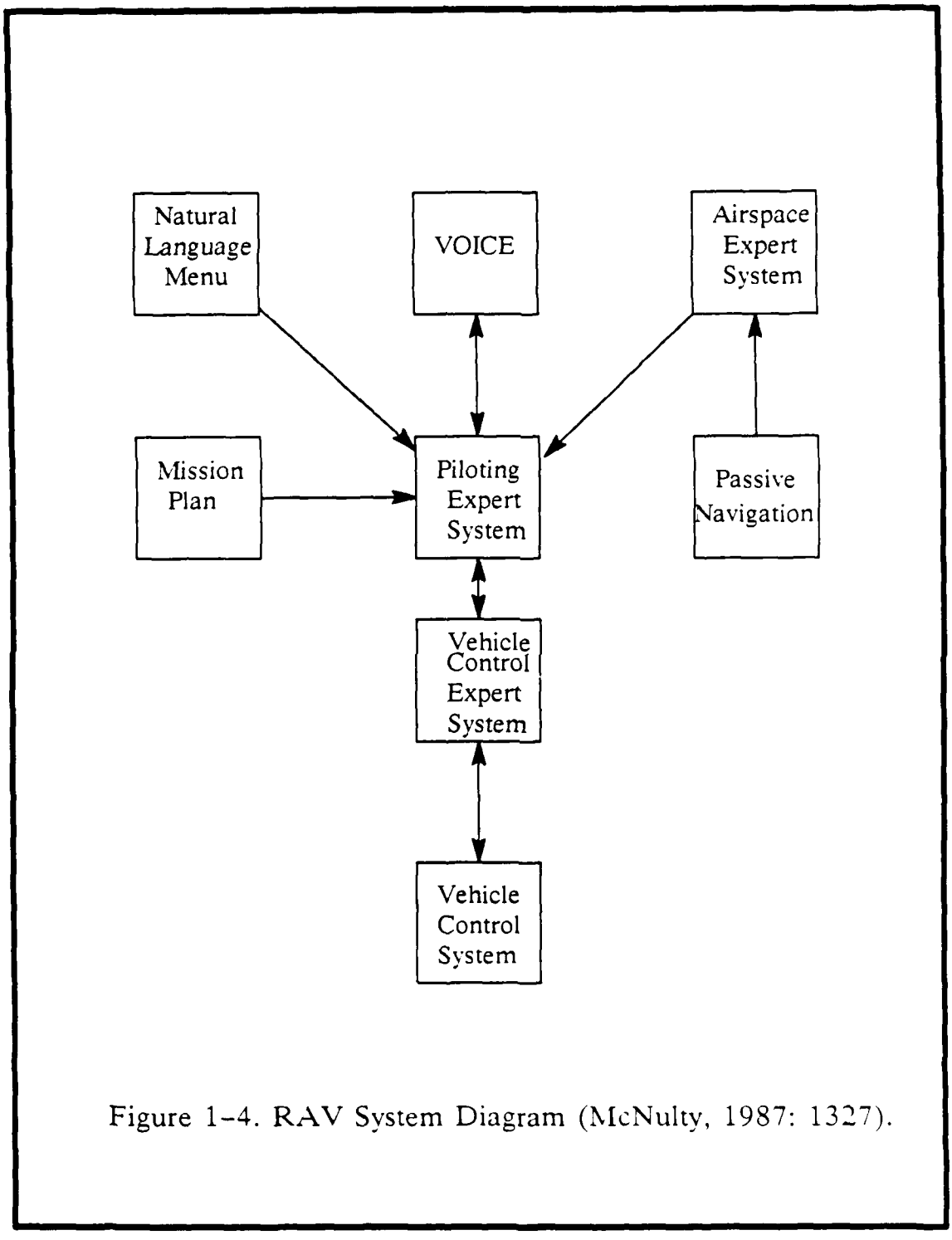


Figure 1-4. RAV System Diagram (McNulty, 1987: 1327).

the reasons for the architectures chosen is contained in Appendix B. The first one is the host of the RAV system. This architecture is a network of four loosely coupled Texas Instruments' Explorer Lisp machines with distributed memory. The other is the Intel iPSC hypercube with up to 32 processing elements (PEs). The results could possibly be applied to other intelligent control expert systems and other expert systems in general on the same architecture type. Some basic results could possibly be generalized to other examples of searching not associated with expert systems and to other expert systems on closely related architectures.

This research is intended to be a feasibility study of the issues involved including implementing an expert system written in Automated Reasoning Tool (ART) on a TI Explorer on the iPSC Hypercube using Concurrent Common LISP (CCLISP). ART is a knowledge engineering language used in the development of expert systems. CCLISP is a dialect of Gold's Common LISP that has been enhanced to allow for message passing on the iPSC Hypercube. LISP was chosen since it was available on both the TI Explorer and iPSC Hypercube making the transportation of the code from one machine to the other easier. This study is most interested in examining the execution speed of real-time systems that use production systems. This study focuses on achieving execution "speedup" through the use of parallel algorithms. The performance metrics of this study will be defined latter in the report. The results of this study are not intended to specify final real-time execution times, but rather an analysis of speedup possibility due to parallel processing of production systems.

Approach

The current knowledge base (Appendix A) for the RAV has been obtained from TI through the Air Force Avionics Laboratory. This includes a basic demonstration routine. Portions of this demonstration will be used to exercise the system. The control for the expert system is developed using the basic principles of production system control for

an inference engine (Appendix E). The current RAV software uses the Automated Reasoning Tool (ART) as the inference engine (McNulty, 1987). ART can not be used with the parallel environment since it is not licensed for nor is available for the iPSC hypercube. The inference engine will be designed on the TI Explorer Lisp machines where it can be tested against the knowledge base and the rule execution timing results can be compared to the ART inference engine. The parallel expert system is implemented on the Intel iPSC hypercube with up to 32 processing elements (PEs) to explore larger degrees of parallelism.

Overview of the Thesis

The thesis is organized into six chapters with detailed explanations and descriptions in various appendices to improve understandability. The first chapter is an introduction of the issues involved with the thesis investigation along with a problem statement, the scope of the study and the approach used in the study. The second chapter examines the key issue of search. The third chapter is a discussion of parallel decomposition of search and how it applies to production systems and the RAV system. The fourth chapter contains a detailed analysis and design of the serial inference engine used, the parallel inference engine, a parallel production system, and a parallel RAV expert system. The fifth chapter provides the implementation details for the TI Explorer system and the Intel iPSC hypercube along with the experimental results. The sixth chapter provides a summary of the results along with conclusions and recommendations of this study. Detailed discussions of the RAV, parallel computer architectures, NP-completeness, and inference engines as well as the serial and parallel production system code are provided as appendices.

II. Searching Knowledge Based Systems

Introduction

Search is a basic method for problem solving where other direct methods (algorithms) do not exist. Problem solving using search involves defining a state space and systematically checking those states to find a solution or solution path. The state space (problem space) forms a graph with the nodes being states and the edges are the transitions from one state to the next state. A solution to the problem is then determined by selecting an initial state (node) and methodically traversing the branches (transitions) to find a goal state (node).

Graphical Representation

The state space graph in its most general form is an AND/OR graph. An example of an AND/OR graph can be seen in Figure 2-1. The successors of a state (node) can be independent of each other (OR connections). The successors of a state (node) can also be related (AND connections) represented by arcs between the edges in the graph. This latter type of relationship indicates that any solution path through one of the nodes in an AND connection must include all the other nodes in the AND connection.

A quick look at some terminology. Once a node has been selected the process of producing its successors is known as expanding a node. The successors are sometimes known as children with the original node being called the parent. In this way, grandparents and grandchildren can be defined and can be useful in describing relationships between nodes on different levels of the graph (Figure 2-2).

Search Types

The two most basic types of search are breadth-first and depth-first. They are both uninformed searches. That is, neither uses any heuristic information to guide the search process. Breadth-first search can be seen in Figure 2-3. The search visits all the

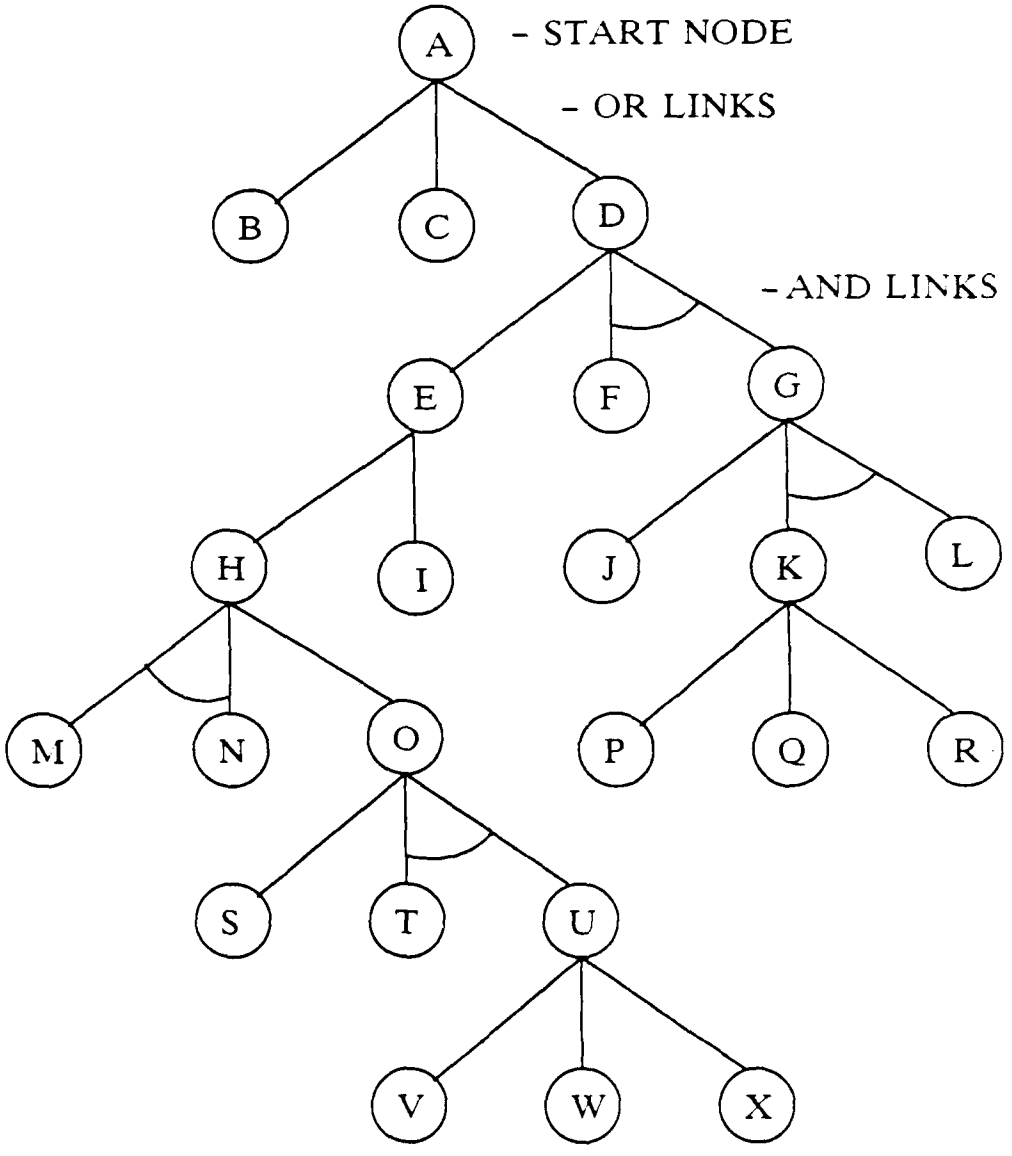


Figure 2-1. AND/OR Tree.

nodes on each level before moving to the next level. This can be viewed as a Last-In First-Out (LIFO) process. Depth-first search can be seen in Figure 2-4. This search visits one node on each level along a vertical path. This can be viewed as a First-In First-Out (FIFO) process.

Before discussing other types of searches, a discussion of solution quality is needed. The term optimality can have various meanings depending on the characteristics of the problem space. If only one goal state exists, then there is no difference in the goal that is found, only the paths to the goal are different. The "best" path to this single goal can be defined in many ways. The "best" path could be the shortest path in terms of the number of nodes on the path. The "best" path could be the minimum cost of the sum of weights of each node on the path. In general, the "best" or "optimal" path can be defined in any consistent manner for any specific problem space. If more than one goal exists, then the "optimal" solution not only involves finding the "best" path to a goal, but also in finding the "optimal" goal. This implies that there is some criteria to order the various goal states, so that one goal can be found to be the "best" or "optimal".

Searches that discover the "optimal" solution path, that is find the "best" path to the "best" goal are known as optimizing searches. They find the "best" answer. Searches that sacrifice the optimality requirement in order to discover a solution are known as satisficing searches. Here the requirement of finding a solution is satisfied, but the optimality of the solution is sacrificed.

There are many types of informed searches. An informed search, unlike breadth-first or depth-first searches, use some knowledge about the problem space to try to reduce the number of nodes examined in the search process. These searches are known as best-first searches. The nodes of the graph are ordered by some criteria. This criteria is defined by what is meant by "best." As was seen above, this can take on various meanings depending on the problem and the situation. The important point is that the nodes are ordered and the first in the ordering is selected for expansion. A* is an exam-

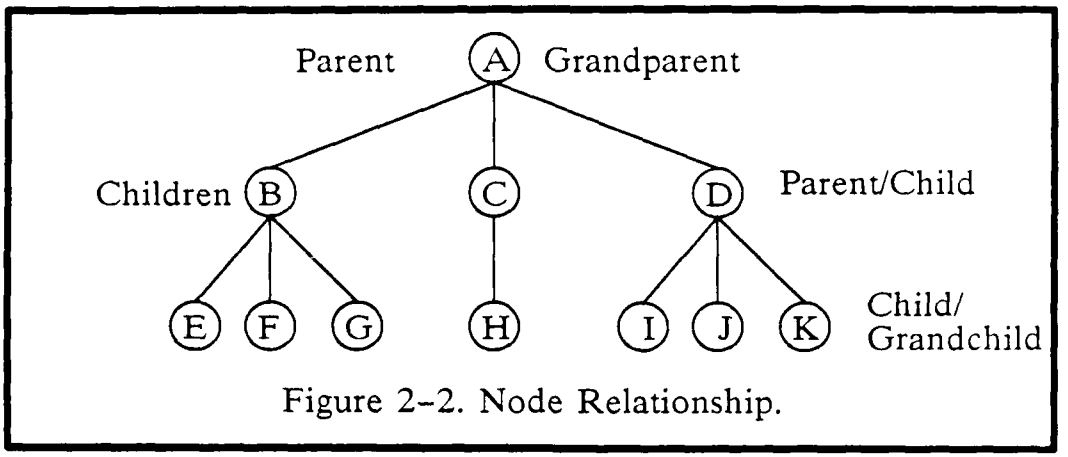


Figure 2-2. Node Relationship.

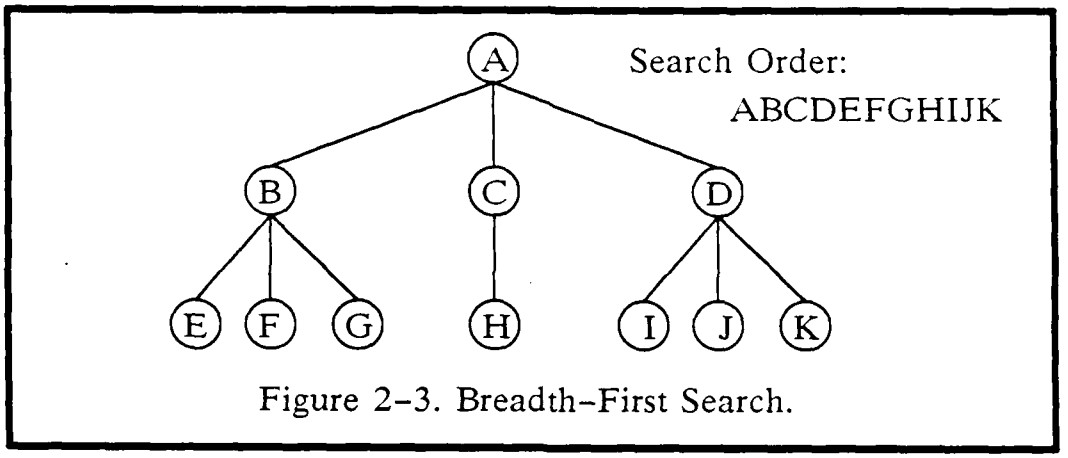


Figure 2-3. Breadth-First Search.

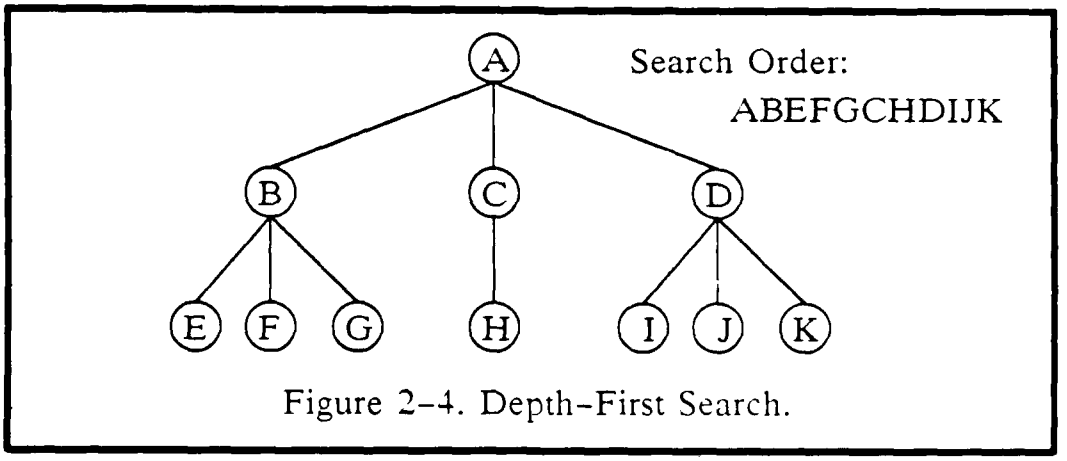
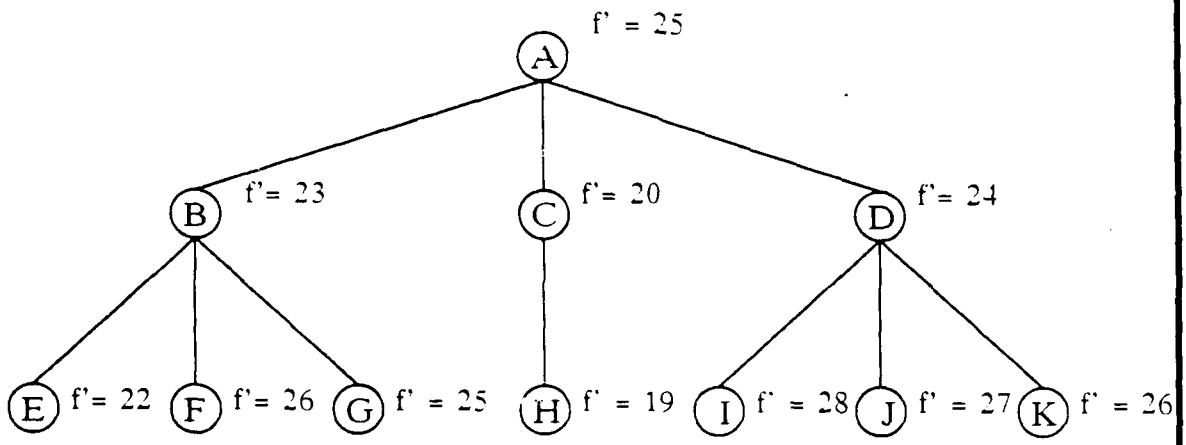


Figure 2-4. Depth-First Search.

ple of this type of search. A* defines the "best"node in terms of a function f' where f' = g + h'. The value of g is the "cost" of the path from the initial node to the current node in question. The value of h' is an estimate of the "cost" of the path from the current node in question to the goal node. The node with the smallest f' value is selected from all the nodes in consideration to be expanded on each cycle of the search. Figure 2-5 shows an example of an A* search. The h' value is called the heuristic and varies from one problem to the next. The optimality of the A* search can be analyzed based on the character-



Search Order:

ACHBEDGFKJI

Figure 2-5. A* Search.

istics of the f' function. If f', the estimate of the cost of the path from the initial node to the goal node through the current node, is less than f, the actual cost, at each node in the graph, then the A* search will find the "optimal" solution based on the f' function.

The other types of best-first search are variations on this basic theme. One type of variation is to change the f' function. This function can be changed to decrease the

number of nodes examined, but at the possible expense of "optimality." This type of change usually involves weighting either the g or h' value or both. Another variation is to eliminate or limit the nodes that have been expanded from being considered in the future of the search. This is usually accomplished by creating a bound where all nodes whose f' value are beyond the bound are eliminated from consideration. This has the effect of trimming the graph and reducing the nodes to be examined. This method, also, speeds the search at the expense of "optimality" and in some cases at the expense of arriving at a solution. Examples of these variations can be found in Figure 2-6 and are discussed in the literature (Nilsson, 1980; Pearl, 1984; Rich, 1983).

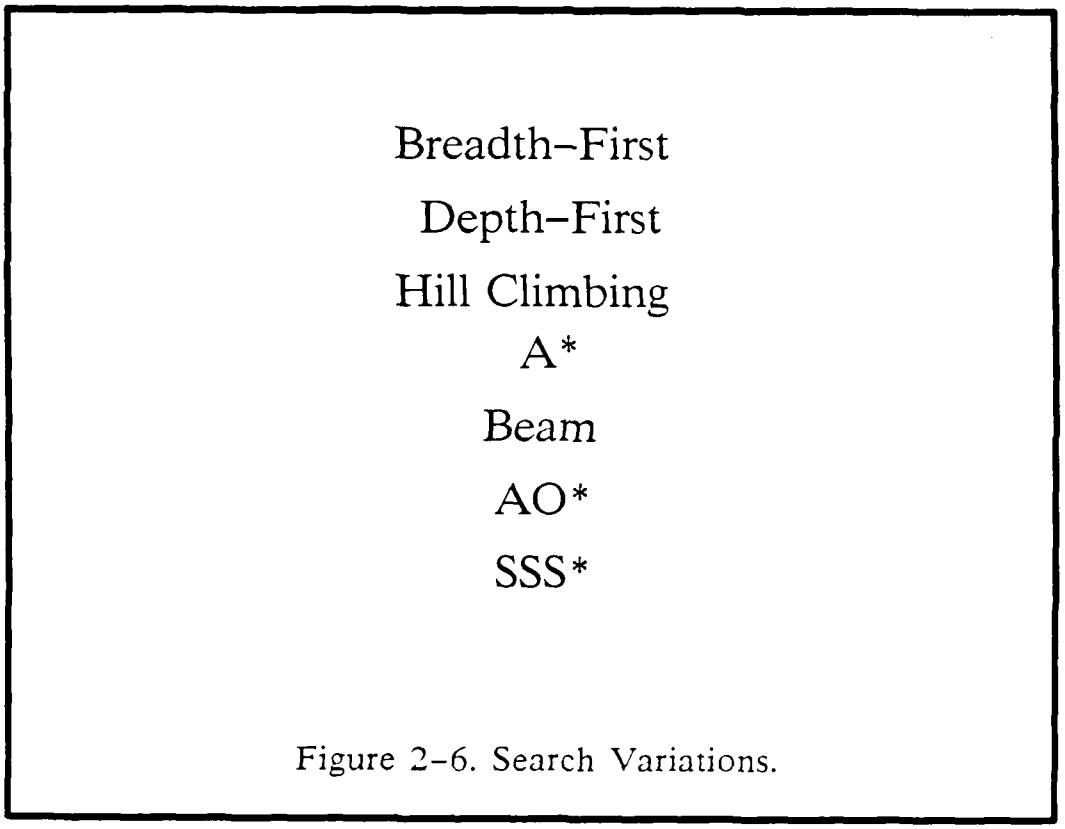


Figure 2-6. Search Variations.

The various types of searches are slight changes to the same form. They all exhibit the same flow. It is a form of generate-and-test or branch-and-bound. The algorithm in Figure 2-7 describes the basic procedure.

1. Select a node from the list of nodes to be explored.
2. Test the node to determine if it is the goal.
3. Generate the children (successors) of the node.
4. Place the children on the list of nodes to be explored.

Figure 2–7. Search Cycle.

The domains of the problem space, therefore, influence the basic operation of the search. The only impact the problem space has on the operation of the search is in the heuristics used to select the next node for expansion. For example, with the typically numerical domains associated with the Traveling Salesman Optimization Problem and the Set Covering Optimization Problem the heuristic is calculated using $f' = g + h'$. In the case of the Traveling Salesman Optimization Problem, the value of h' can be calculated from the weight of the minimum spanning tree associated with the remaining unused nodes. With problems involving a more symbolic logical representation, the heuristics can take a variety of forms. The selection of nodes to expand can be based on the number of terms in the symbolic representation. The selection can be based on the depth of the parents of the node in consideration. In the set-of-support strategy of resolution, the clauses that result from the goal and their descendants are given preference (Nilsson, 1980).

The basic components of search can be found in logical resolution systems and production systems as well as in many branch-and-bound search applications. The only variation is in the optimization requirement and in the heuristics used in selecting the next node to be expanded.

Production Systems and Search

Production systems are computational models. They consist of three components: a global database of facts, a set of productions or rules, and a control strategy. The global database or working memory (WM) contains the set of initial facts or axioms as well as facts generated by the productions. The productions or rules, known as the production memory (PM) consist of an antecedent or a left-hand side (LHS) and a consequent or right-hand side (RHS). The rule is of the form 'IF condition THEN action' where the condition is the LHS and the action is the RHS. The LHS can be any conjunction of facts or negated facts. The RHS can be any new fact or the retraction of an existing fact. The control strategy, also known as an inference engine, matches the LHS of the rules of the production memory with the facts in the working memory. The inference engine maintains a list of bindings of variables with possible matches from the working memory. After all the rules have been matched, the inference engine selects, from all the rules that have successfully matched, a rule to apply. The RHS of this rule is then applied resulting in the addition or subtraction of facts from the working memory. The basic cycle can be seen in Figure 2-8.

As a computational model, Post proved in 1943 that a formal production system is computationally equivalent to a Turing machine (Post, 1943; Minsky, 1967). This means that any Turing computable function can be expressed as a production system. This gives production systems a great deal of computational power. This implies, however, that production systems do suffer from the halting problem. The halting problem occurs when a solution fails to exist, then the production system is not guaranteed to halt, but can run forever.

The advantages in using a production system lies in its flexibility. The knowledge is separated from the logic or control. This allows the rules to be changed without

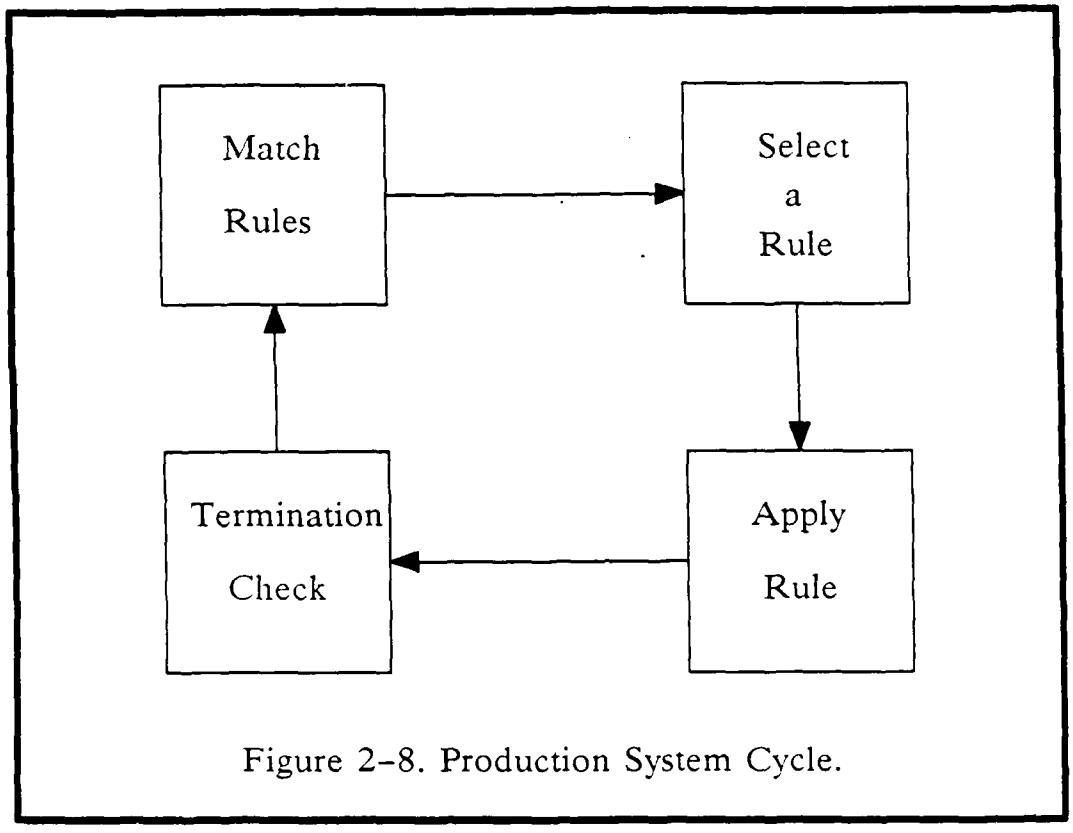


Figure 2-8. Production System Cycle.

affecting the control logic of the whole system. This gives the system designer greater flexibility in system growth (Nilsson, 1980).

The control of a production system is essentially a search process. The current state of working memory elements or facts can be represented as nodes. The rules are the transitions to a new state. The number of rules that can be applied in any one situation is the branching factor of the graph at that node or state. The basic control algorithm is seen in Figure 2-10. A graphical depiction of a production system as a search graph is seen in Figure 2-9. OR branches of the graph occur when rule changes the working memory is such a way so that another rule that did apply, no longer applies. AND branches of the graph occur when the application of one rule does not impact application of another rule.

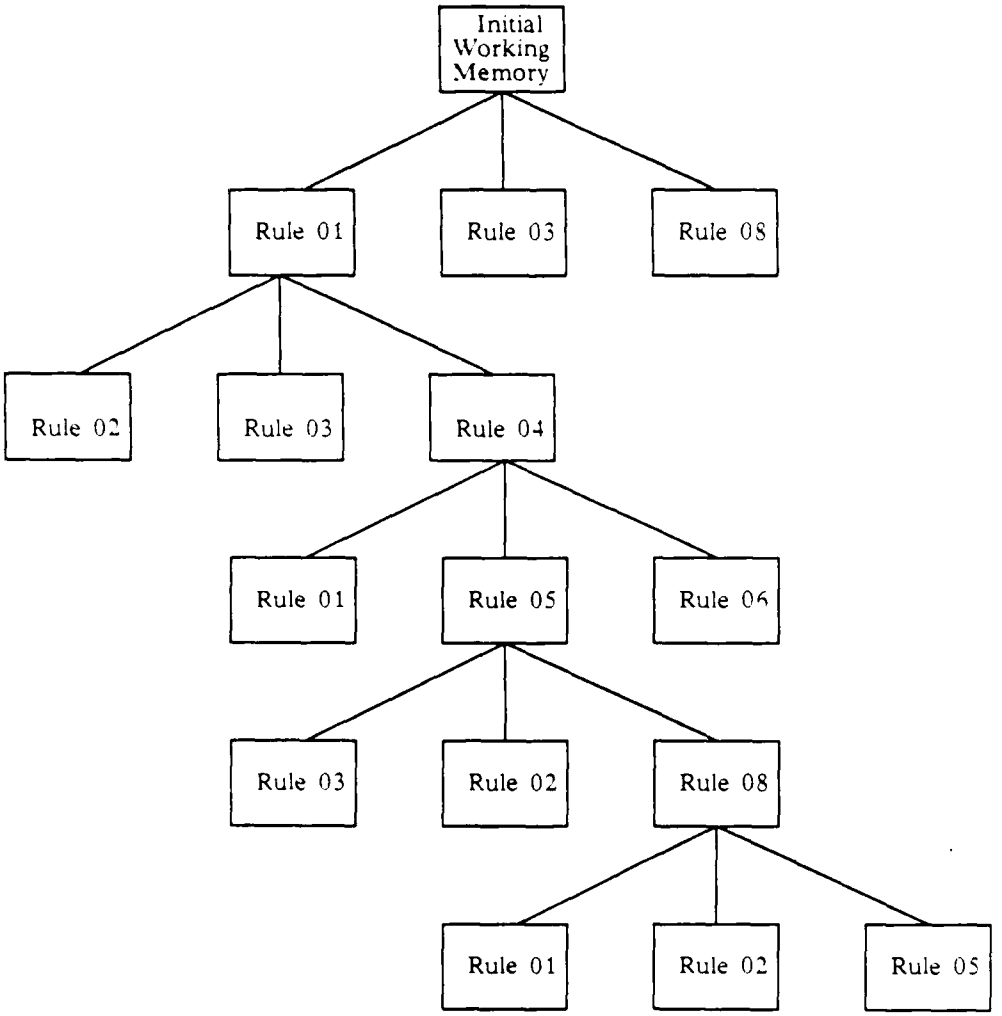


Figure 2–9. Production System Graph.

Procedure PRODUCTION

1. DATA <- initial database
2. until DATA satisfies the termination condition, do;
3. begin
4. select some rule, R, in the set of rules that can be applied to DATA
5. DATA <- result or applying R to DATA
6. end.

Figure 2-10. Production System Algorithm (Nilsson, 19080: 21).

The algorithm in Figure 2-10 is nondeterministic. In step 4, several rules can be selected, but on a serial architecture only one rule can be applied in step 5. Due to the OR branches in the graph and the nature of a production system, once certain rules have been applied there is no way to reverse the effect and to backtrack in the graph. This is an irrevocable control strategy. This is especially true of a production system operating in a real-time environment where performance is critical. However, another characteristic of a production system that can overcome an irrevocable control strategy is the commutative property. This means that any action can be gotten from any ordering of rules (Nilsson, 1980). AND branches offer one example of this type of commutative nature to production systems.

Summary

This chapter reviewed search and production systems. Production systems are a very expressive computational model in that all other models can be represented by a

production system. This chapter showed that a production system can be viewed as a search process. In the next chapter, methods are developed to arrive at parallel algorithms for production systems. These algorithms are then used to design and implement a concurrent production system.

III. Parallel Search Algorithms

Introduction

This chapter looks at parallel processing of search algorithms. First, a look at parallel decomposition and performance metrics associated with parallel computing. This chapter explores the different manisfestations of parallelism within general search and specifically the concurrency available in production systems. Finally in this chapter, various algorithms are presented for parallel decomposition of a production system to lay the groundwork for the design in chapter four.

Parallel Decomposition

The advent of parallel computer architectures have brought about the possibility of faster execution of many computer applications. Parallel architectures have brought about new problems as well as the old in terms of software analysis and design. For an application to be implemented on parallel architecture, a way must be found to decompose the problem into component pieces. Several important issues are concerned with this decomposition. First, the work must be distributed as evenly as possible. This is so that each processor is busy. This is load balancing. Second, the communication between the pieces needs to be kept to a minimum. This is to reduce the communication overhead associated with the various processors communicating with each other. However, when this communication occurs, the processors need to be synchronized with respect to each other. This is to prevent problems with updating shared variables that can produce erroneous or unpredictable results. Proper synchronization also prevents the occurrence of deadlock between processors (Ishida and Stolfo, 1985).

Performance Measurements

Speedup is the most common performance measurement (metric) in parallel computing. This is the ratio of the run time of the concurrent software running on n nodes

over the the run time of the best serial solution. An application is said to be "perfectly parallel" if this ratio is n. This is a linear speedup. That is the speedup goes up linearly with the number of processors. Often the speedup approaches the linear speedup, but does not reach it due to the communications overhead between the processing elements (Gupta, 1986). Although rare, speedups have been observed greater than n. This is called super linear speedup. At first this seems to be absurd, but upon further study it does seem reasonable. Super linear speedup usually occurs when the application is so large on a serial system that certain overheads are incurred, but when placed on many processors none of the pieces is large enough to incur the same overhead. Thus super linear speedup is observed(Kornfeld, 1981). In addition, there is usually a point at which the addition of more processors does not improve the speedup (Gupta, 1986).

Communications overhead is a large concern is parallel computing. This communication takes several forms. The first is the time to set the job up on the parallel system or the time to distribute the work. The second involves the time needed to collect the results of the job. The third is the communication needed between the processors during the running of the job. An important measurement is the time a processor is communicating versus processing. This measurement along with the setup time and cleanup time gives a good indication of the overhead associated with the parallel process. This is not the only area which produces overhead within the parallel process.

Load balancing is yet another important criteria for parallel computing. This is the percentage of the total processor power that is used during the job. A perfect load balance would be one in which all the processors are busy all the time. This perfect balance is impossible due to two factors. First, depending on the connection network for the processors (see Appendix B) the setup and cleanup provide for times when not all the processors are busy. Second, there is usually some fraction of the job that is inherently serial. This part of the job has to be performed on one processor while the other proces

sors are idle. These two factors are innate to the problem. Poor load balancing can also be designed into a problem due to a poor decomposition.

Several other performance measurements are needed to baseline a production system. These include 1) the number of productions or rules, 2) the number of working memory elements or facts, 3) the composition of the rules which includes the number of clauses in the LHS and RHS of the rule, and 4) the average number of rules eligible to be selected on a given cycle. These are but a few basic components, other characteristics depend on the system and inference engine being examined.

Parallel Search

The general forms of the search were discussed in chapter two. Now, the issue is how to decompose the problem to perform the search in parallel. Like any decomposition problem, there are two main choices for concurrency. Either decompose the functions or instructions and perform independent tasks concurrently or divide the data and perform the same functions on subsets of the domain. In some cases, both can be achieved in layers. For search, both methods have been proposed. For example, Mraz studied parallel branch-and-bound search by dividing the operations of a search cycle among different processors (Mraz, 1986). Gupta studied the decomposition of the data within production systems. In his study, Gupta looked at generating the next nodes in the search graph through parallel matching of rules using the Rete algorithm (see Appendix E for more details on Rete) (Gupta, 1986).

Both of these methods can be mapped onto the two main categories of parallel architectures. The functional decomposition can be mapped onto the MIMD architecture while the data decomposition can be mapped onto the SIMD architecture. However, problems that can be mapped to the SIMD machine can also be implemented on the MIMD machine. This makes the MIMD architecture suitable for either decomposition as well as the layered decomposition. The layered decomposition involves decomposing the rules on

one level and within that level decomposing the facts. Consider a case where the rules are distributed among several subnetworks of PEs. Then within each subnetwork each the data is divided into independent groups to match against the set of rules given to that subnetwork. This provides a layering affect of decomposition.

The decomposition of the functions is straight forward with a limited possibility for parallelism. The cycle can be seen to be a four step process (Figure 3-1). If this

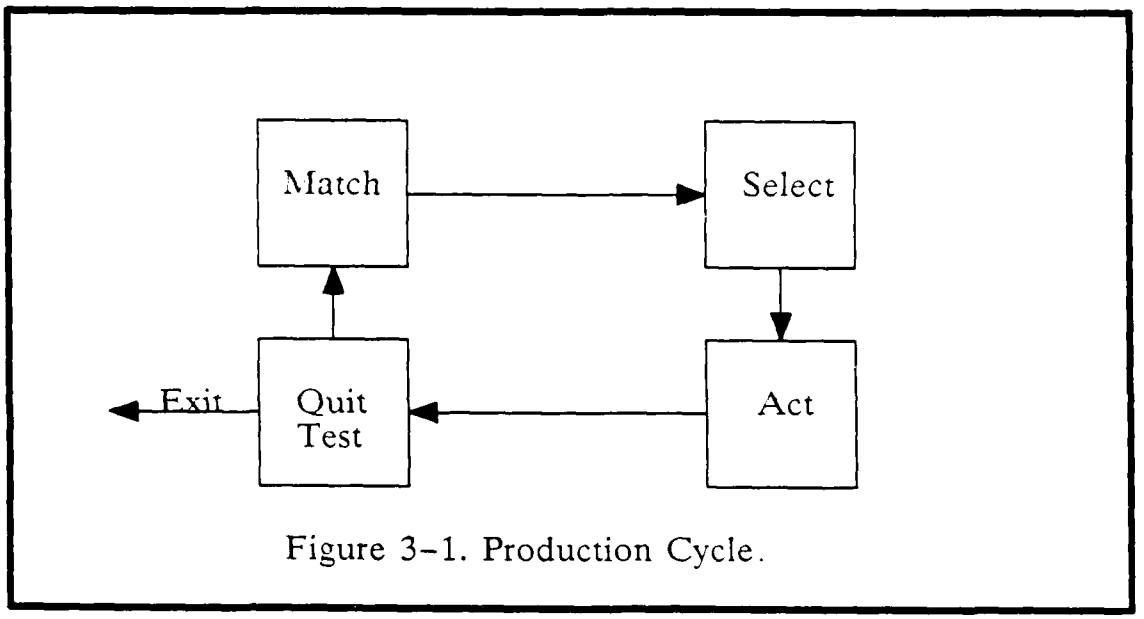


Figure 3-1. Production Cycle.

decomposition could be implemented in a "perfectly parallel" fashion, then only a four times speedup could be achieved. Greater speedups are hoped for. This can only be achieved by analyzing the data.

The data decomposition offers a much larger opportunity for concurrency. This is due to large set of data. If the search graph or tree can be decomposed then there is the potential for great speedups. The question is how to decompose an exponentially growing graph or tree among a constant number of processors. This decomposition has to be

constrained in such a way as to insure a "fairly equitable" load balancing and to reduce the communication due to data dependencies.

Parallel Production System

Production systems, as were seen in chapter two, are a type type of search. Therefore, parallel decomposition techniques for search problems can be applied to production systems. So, production systems can be decomposed along the control functions like a branch-and-bound or it can be decomposed by its data. In the case of a production system the data can be thought of as two parts (Figure 3-2). The first part is the facts or the working memory (WM). Although the working memory can have several meanings in this context it refers to the initial facts and axioms as well as the facts added due to the firing of rules. The second part is the rules or the production memory (PM). These two parts are not always distinct, but can overlap. For example, the result of a rule could be the addition of a new rule. The reasons for making the distinction in the types of data is that in some cases it is much easier to decompose the rules than it is to decompose the facts. The latter requires data dependencies to be worked out while the former requires less restrictive decomposition considerations. Both types of decompositions will be looked at in the algorithms.

As described earlier the concurrency available in decomposing the functions is limited. This is particular true for production systems where over 90% of the time is spent in the match function (Gupta, 1986). So the main emphasis is placed on the decomposition of the data. The methods for implementing a production system tend to center around ways to decompose the rules (PM) and the facts (WM). This has lead to several algorithms to accomplish this decomposition and placement on separate processors. These algorithms are generally at the level where the underlying inference engine structure is unimportant. The methods are more concerned with the dependencies of the rules on each other and the facts (WM).

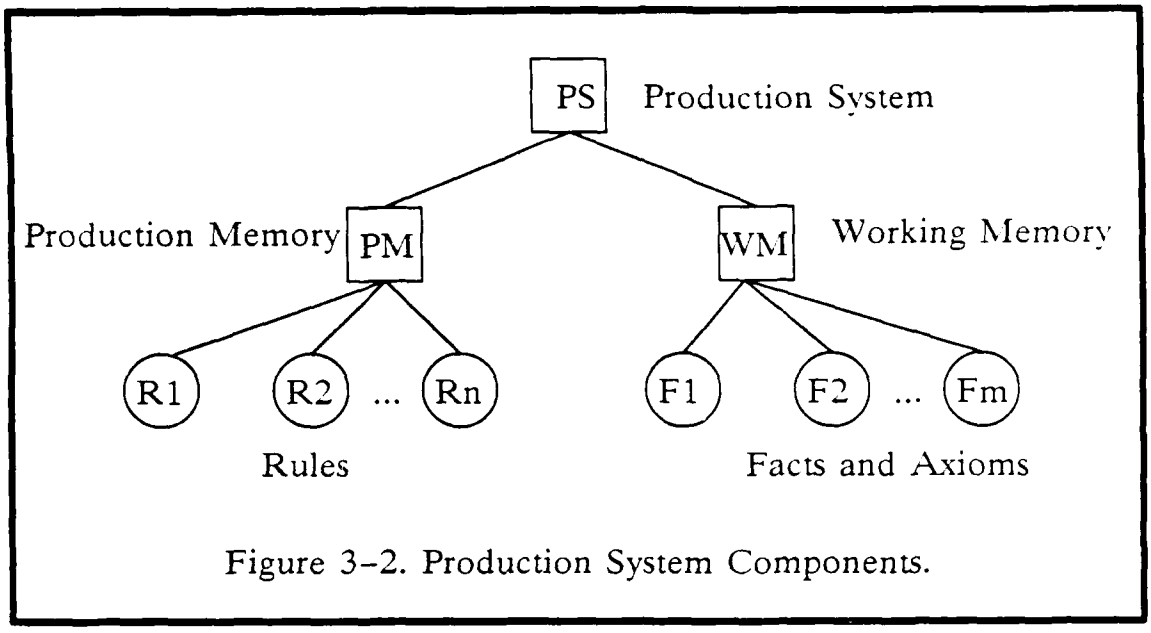


Figure 3-2. Production System Components.

Algorithms

Several algorithms have been proposed for the parallel decomposition of production systems. These algorithms deal with rule independence and communication among processors. The first five algorithms (Stolfo, 1984) are for the DADO parallel architecture. Although some aspects of these algorithms are hardware dependent, they can be generalized to other architectures as well. The DADO architecture consists of a binary tree of processors which is easily simulated on an n-cube architecture like the iPSC hypercube. One hardware dependent feature is a process called max-resolving that is specialized hardware to select the maximal PE for selection of a rule to fire. The DADO PEs have the capability to alternate between MIMD mode and SIMD mode (Stolfo, 1984). The last two algorithms in this chapter provide specific algorithms for rule decomposition and rule synchronization. This is accomplished using data dependency graphs for production systems.

Algorithm 1 - Full Distribution of Rules

This algorithm distributes all the rules across all the available PEs. Each PE has its own copy of the WM. This algorithm can be seen in Figure 3-3.

1. Initialize: Distribute a simple rule matcher to each PE. Distribute a few distinct rules to each PE. Set CHANGES to initial WM elements.
2. Repeat the following:
3. Act: For each WM-change in CHANGES do
   a. Broadcast WM-change (add or delete a specific WM element) to all PE's.
   b. Broadcast a command to locally match [Each PE operates independently in MIMD mode and modifies its local WM. If this is a deletion, it checks its local conflict set and removes rule instances as appropriate. If this is an addition, it matches its set of rules and modifies its local conflict set accordingly].
   c. end do;
4. Find local maxima: Broadcast an instruction to each PE to rate its local matching instances according to some predefined criteria.
5. Select: Using the high-speed max--RESOLVE circuit of DADO2, identify a single rule for execution from among all PE's with active results.
6. Instantiate: Report the instantiated RHS actions. Set CHANGES to the reported WM-changes.
7. end Repeat;

Figure 3-3. Algorithm 1 (Stolfo, 1984: 302).

This algorithm is rather straight forward to implement since all of the working memory is located at each PE. This could be a limitation if the WM is too large to fit on a

PE. The speed of this algorithm depends on the speed of the match on each PE. Multiple firing of rules can be accomplished, but is handled in detail in a later algorithm (Stolfo, 1984). This algorithm is the basis for the algorithm developed in chapter four.

Algorithm 2 - Original DADO Algorithm

This algorithm divides the DADO machine into three sets of levels: a PM-level, an upper tree, and WM subtrees. The upper tree is the meta control level to supervise operation of the system. The PM-level and WM-subtrees form a layered decomposition of the production system. The PM-level distributes the rules and then each of these subsets of rules are matched in parallel by the WM-subtrees (Stolfo, 1984). This algorithm can be seen in Figure 3-4.

This algorithm is very dependent on the DADO architecture. This algorithm also forces a limitation on the size of the PM, but the size of the WM can be quite large. For these reasons this algorithm, does not weigh in very heavily on the design of the parallel inference engine in chapter four. This algorithm starts to approach the idea of a distributed WM that allows for multiple rule firings.

Algorithm 3 - Miranker's TREAT Algorithm

This algorithm was developed by Daniel Miranker to improve on algorithm 2 to provide some features of the Rete algorithm. This algorithm can be seen in Figure 3-5.

This algorithm is appropriate for production systems that have small PMs and large WM and where many rules are affected on each cycle. This algorithm uses a simple matcher rather than the Rete algorithm (Stolfo, 1984). This fact makes this a rather interesting algorithm for incorporation in chapter four.

Algorithm 4 - Fine Grain RETE Algorithm

This algorithm uses a complete compiled Rete network on each PE. Otherwise, this algorithm is very much like algorithm 1. This algorithm can be seen in Figure 3-6.

1. Initialize: Distribute a match routine and a partitioned subset of rules to each PM-level PE. Set Changes to the initial WM elements.
2. Repeat the following:
3. Act: For each WM-change in CHANGES do;
   a. Broadcast WM-change to the PM-level PE's and an instruction to match.
   b. The match phase is initiated in each PM-level PE:
      i. Each PM-level PE determines if WM-change is relevant to its local set of rules by a partial match routine. If so, its WM-subtree is updated accordingly. [If this is a deletion, an associative probe is performed on the element (relational selection) and any matching instances are deleted. If this is an addition, a free WM-subtree PE is identified, and the element is added.]
      ii. Each pattern element of the rules stored at a PM-level PE is broadcast to the WM-subtree below for matching. Any variable bindings that occur are reported sequentially to the PM-level PE for matching of subsequent pattern elements (relational equi-join).
      iii. A local conflict set of rules is formed and stored along with a priority rating in a distributed manner within the WM-subtree.
   c. end do;
4. Upon termination of the match operation, the PM-level PE's synchronize with the upper tree.
5. Select: The max-RESOLVE circuit is used to identify the maximally rated conflict set instance.
6. Report the instantiated RHS of the winning instance of the root of DADO.
7. Set changes to the reported action specifications.
8. End repeat;

Figure 3-4. Algorithm 2 (Stolfo, 1984: 303).

This works best on a production system where the PM is large with relatively few rules affected on each cycle and the WM is small (Stolfo, 1984). Due to the use of the

1. Initialize: Distribute to each PM-level PE a simple matcher (described below) and a compiled set of rules. Distribute to the WM-subtree PE's the appropriate pattern elements appearing in the LHS of the rules appearing in the root PM-level PE. Set CHANGES to the initial WM elements.
2. Repeat the following:
3. Act: For each WM-change in CHANGES do;
   a. Broadcast WM-change to the WM-subtree PE's.
   b. If this change is a deletion, broadcast an instruction to match and delete WM elements and any affected conflict set instances calculated on previous cycles.
   c. Broadcast an instruction to PM-level PE to enter the Match Phase.
   d. At each PM-level PE do;
      i. Broadcast to WM-subtree PE's an instruction to match the WM-change against the local pattern element.
      ii. Report the affected rules and store in L.
      iii. Order the pattern elements of the rules in L appropriately.
      iv. For each rule in L do;
         1. Match remaining patterns of the rules specified in L as in Algorithm 2.
         2. For each new instance found, store in WM-subtree with a priority rating.
         3. end do;
      v. end do;
   e. end for each;
4. Select: Use max-RESOLVE to find the maximally rated instance in the tree.
5. Report the winning instance.
6. Set Changes to the instantiated RHS of the winning rule instance.
7. end Repeat;

Figure 3-5. Algorithm 3 (Stolfo, 1984: 305).

1. Initialize: Map and load the compiled Rete network on the DADO tree. Each node is provided with the appropriate match code and network information. Set CHANGES to initial WM elements.
2. Repeat the following:
3. Act: For each WM-changes in CHANGES do;
   a. Broadcast WM-change (a Rete token) to the DADO leaf PE's.
   b. Broadcast an instruction to all PE's to Match. (First, the leaf processors execute their one-input test sequences on the new token. The interior nodes lay idle waiting for match results computed by their descendants. Those tokens passing the one-input tests are communicated to the immediate ancestors which immediately begin processing their two-input tests. The process in then repeated until the physical root of DADO reports changes to the conflict set maintained in the DADO control processor).
   c. end do;

Select: The root PE is provided with the chosen instance from the control processor. Set CHANGES to the instantiated RHS.
4. end Repeat;

Figure 3-6. Algorithm 4 (Stolfo, 1984: 306).

Rete algorithm and these performance characteristics it is not considered for the parallel inference engine in chapter four, but is provided for a sense of completeness.

Algorithm 5 - Multiple Asynchronous Execution

This algorithm addresses the issue of multiple rule firings. This is done by creating multiple root nodes within the tree where rule selection takes place. This algorithm can be seen in Figure 3-7.

1. Initialize: Logically divide DADO to incorporate a static Production System-Level (PS-level), similiar to the PM-level of Algorithm 2. Distribute the appropriate PS program to each of the PE's at thePS-level.
2. Broadcast an instruction to each PS-level PE to begin execution in MIMD mode. (Upon completion of their respective programs, each PS-level PE reconnects to the tree above in SIMD mode.)
3. Repeat the following.
   a. Test if all PS-level PE's are in SIMD mode.
   End Repeat;
4. Execution complete. Halt.

Figure 3-7. Algorithm 5 (Stolfo, 1984: 306).

This algorithm is very flexible, but depends on the rule independence as is discussed in algorithm 7. The principles from this algorithm are used in the design of the parallel inference engine to provide parallel rule firings.

Algorithm 6 - Rule Decomposition

This algorithm deals with the decomposition of rules and the allocation of processors. This process generates a rule tree in such a way that the rules on each processor are independent of each other. The algorithm can be seen in Figure 3-8.

This algorithm is an effective way to divide the rules. However, it is very labor intensive and should be automated. This algorithm assumes no prior knowledge of the production system. With the modular design of the RAV system, the decomposition of the rules should be easier. Each module should be independent.

1. Phase 1: Generating a Rule Tree. A token is defined as a triple, (ruleA ruleB P(ruleA, ruleB)), where rule A and rule B are production rules and P(ruleA, ruleB) is the parallel executibility between rule A and rule B. Parallel executibility is defined between each pair of rules as the number of production cycles which can be reduced by allocating the two rules in distinct PEs. Form a rule tree in which each rule is associated with a distinct leaf node. The goal is to maximize a sum of P(i, j) at each non-leaf node in all combinations of i and j, where i/j indicateds a rule in a right/left subtree of the non-leaf node.

2. Phase 2: Create Partitions for Parallel Processor Systems This phase creates partitions of a production system for parallel processor systems from the rule tree. Because the rule pairs with a large parallel executibility are decomposed in the early stage, partitions for a parallel processor system can be easily obtained by selecting a suitable layer of the rule tree. The tree is binary and thus a single level of the tree is best mapped onto processors containing a number of PEs that is a binary power.

Figure 3-8. Algorithm 6 (Ishida and Stolfo, 1985: 570-571).

Algorithm 7 - Rule Synchronization

This algorithm deals with the synchronization problem of rules. The synchronization of rules can be decided by building a data dependency graph for the production system. The process for building this graph is seen in Figure 3-9.

This algorithm like algorithm 6 is best automated. Again this would be particular useful when the format of the production system is relatively unknown or unorganized. With the organization of the RAV system, a detailed execution is unnecessary. The RAV production system is very independent according by this algorithm's standards.

The process for building the graph is as follows:

1. A production node (P-node) represents a production rule.

2. A working memory node (W-node) represents a group of working memory (WM) elements called a class.

3. A directed edge from a P-node to a W-node represents the fact that the right-hand side (RHS) of a production rule modifies (adds or deletes) a class of WM elements. When a rule adds (deletes) WM elements of a class, the class is called '+' changed ('-' changed), and the corresponding edge is labelled '+' ('-').

4. A directed edge from a W-node to a P-node represents the fact that the left-hand side (LHS) of a production rule refers to a class of WM elements. When a class is referenced by a positive (negative) condition element of a rule, the class is called '+' referenced ('-' referenced) and the corresponding edge is labelled '+' ('-').

Synchronization is required between rule A and rule B if there exists a WM class which satisfies any of the following:

1. '+' changed ('-' changed) by rule A and '-' referenced ('+' referenced) by rule B.

2. '+' changed ('-' changed) by rule B and '-' referenced ('+' referenced) by rule A.

3. '+' changed ('-' changed) by rule A and '-' changed ('+' changed) by rule B.

Figure 3-9. Algorithm 7 (Ishida and Stolfo, 1985: 569-570).

Summary

Most of the algorithms to decompose a production system deal with decomposition of rules over the decomposition of working memory or functions. The decomposition of functions yields little since match is the dominate function that accounts for up to 90% of the processing time (Gupta, 1986). Only the data decomposition of the rules within the match that yield the best hope for concurrency. The decomposition of the working memory requires that the rules break down into independent sets that only operate on a portion of working memory. Algorithm 6 and 7 help to accomplish this, but this requires a great deal of effort. The rules themselves can be decomposed quickly and automatically without regard to what working memory elements they effect.

IV. Analysis and High-Level Design

Introduction

The purpose of this chapter is to present an analysis of the RAV system, along with analysis and design of the serial inference engine, the parallel inference engine, and the parallel RAV production system. The algorithmic design of the parallel RAV production system is based upon the the design of algorithms presented in Chapter three.

RAV Analysis

A detailed description of the RAV is contained in Appendix A. The main portion of the piloting control is a layered series of two expert systems. Each expert system has several components organized by functionality (Figure 4-1). These components provide a source of data independence of rules and working memory. The system contains an "average size" production and working memory. The system contains over 350 rules. The working memory consists of schemas which are a frame-like structure. Each frame contains slots that hold the individual facts. There are approximately 160 schemas. The average number of slots per frame is approximately ten, therefore the total number of facts is about five times the number of rules.

Inference Engine Analysis

The requirement for an inference engine is to perform the basic production system cycle: match, select, and act. [This cycle is also prevalent in the resolution process (Nilsson, 1980).] This inference engine software should be able to match the rules of the RAV expert system with the facts in working memory. It should select one of these rules and add the results of the RHS of the selected rule to the working memory.

The current RAV system implemented on the TI Explorer uses the Automated Reasoning Tool (ART) for this process. ART is licensed software not available on the

RAV Components

PES

Autopilot
Commnav
Departures
Hold-Arc
Intercepts
Intercepts--New
Landings
Mission
Recover
Takeoffs
Targets
Target-All

VCES

Airwork
Airwork-New
Autopilot
Elevation
Heading
Speedbrake
Throttle

Figure 4-1. RAV Components.

iPSC hypercube. The source code for ART is not available, and therefore, can not be modified for parallel execution. This requires that another control process implementation be developed. However, the new implementation should be as compatible as possible with the ART rules and working memory structure as possible.

ART is a very complex and extensive tool. To try to rebuild the generic ART system would require a prohibitive development time. Therefore, simplicity of design is a critical component. The new control process should only provide the functionality of ART that the RAV requires.

Inference Engine Design

The design of the control process had several phases. The first phase was a data flow description of the system as seen in Figure 4-2. This lead to an investigation into the data representation to be used for the working memory and the rules. Three choices were considered. The first involved simply a linked list of rules and a linked list of facts (see Figure 4-3). The second was a linked list of rules and a series of frames for the facts (see Figure 4-4). This was considered due to the structure of the facts as implemented in ART. The facts were implemented as slots within frames or objects. The rules then referenced these slots. In this way, the rules are indexed directly into the fact database by the frame and slot name avoiding a costly serial search of the facts. The third choice considered was a complete Rete network involving the rules and facts (see Figure 4-5). This third choice, although efficient, was very complex and had considerable overhead. Therefore, the second option was chosen due to its simplicity and relative efficiency.

The second phase was a control flow of the inference engine (see Figure 4-6). This control flow was decomposed into its component parts of match, select and act (see Figure 4-7). Once the data representation had been decided upon, the control was relatively straight forward. A skeleton of the design was taken from Winston and Horn

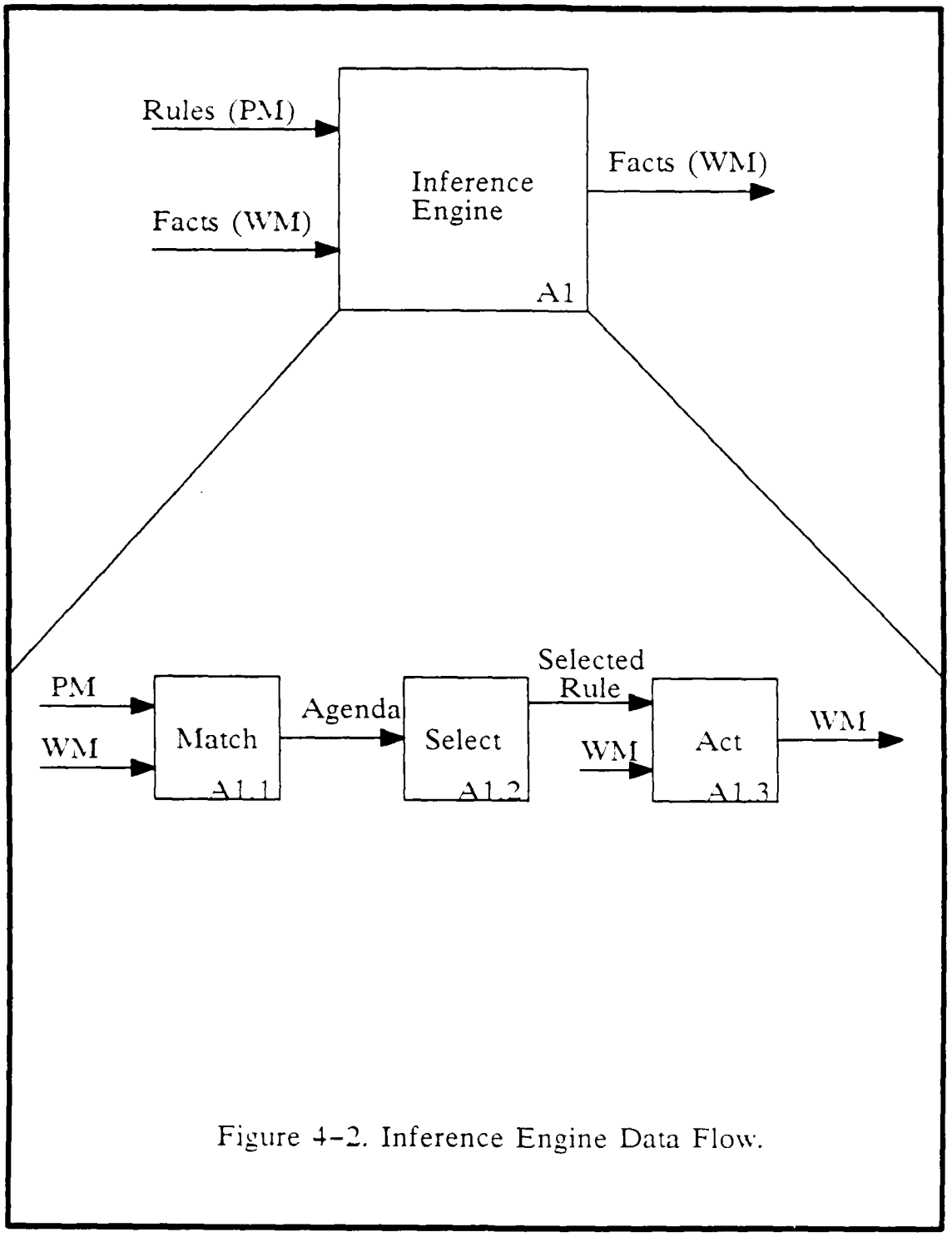


Figure 4-2. Inference Engine Data Flow.

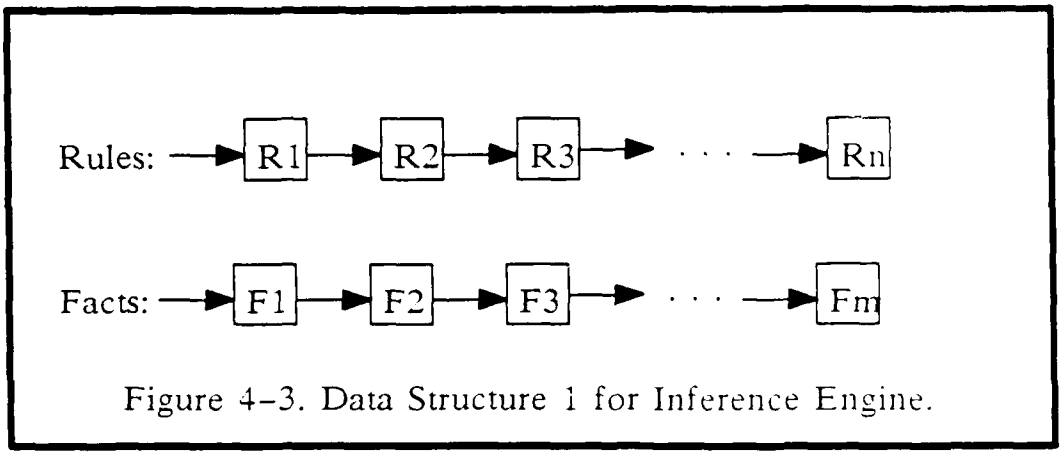


Figure 4-3. Data Structure 1 for Inference Engine.

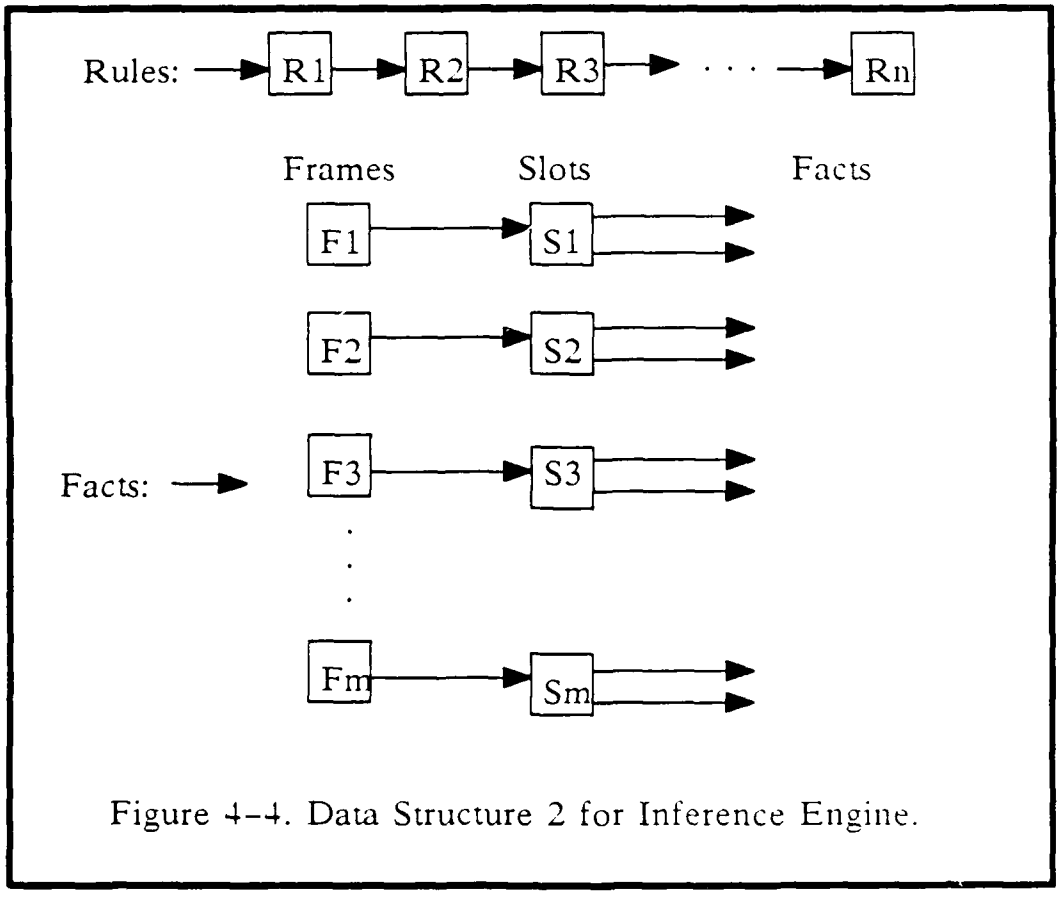


Figure 4-4. Data Structure 2 for Inference Engine.

Rule Conditions:

1) (C1 ^attr1 12 ^attr6 <= 7)
   (C2 ^attr2 > 5)
   (C4)
2) (c1 ^attr1 12 ^attr2 <X>)
   (C3 ^attr3 <X>)
   (C4)
3) (C2 ^attr2 > 5 ^attr3 <Y>)
   (C4 ^attr1 <Y> ^attr3 >= <Y>)

Rete Network:

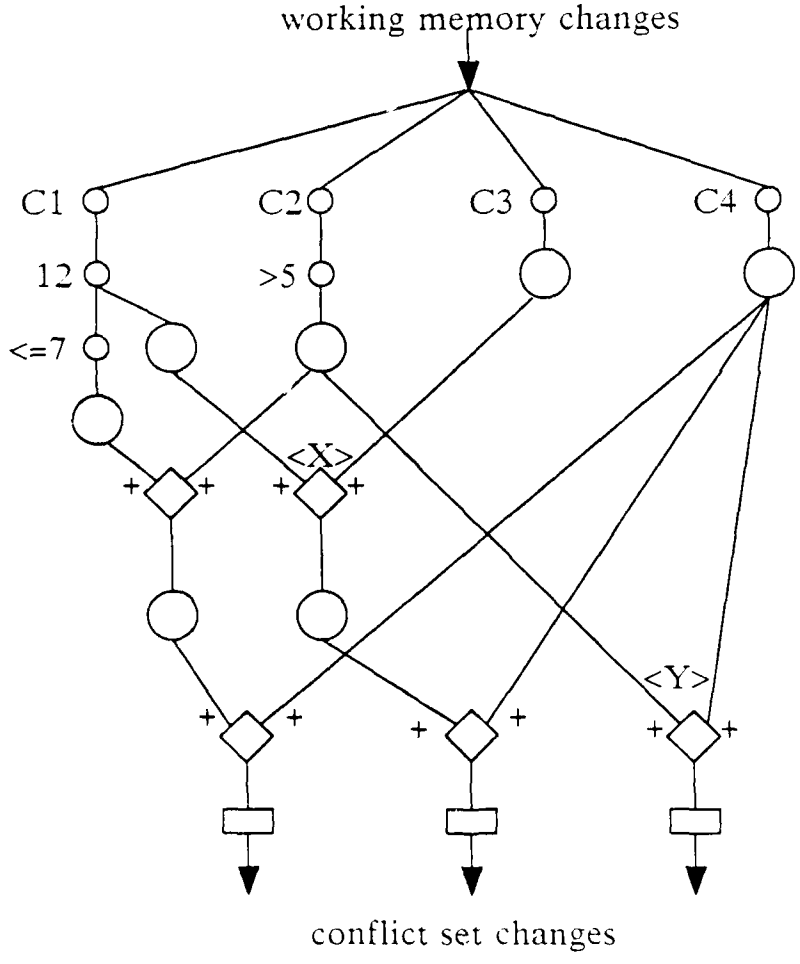


Figure 4-5. Data Structure 3 for Inference Engine (Rete). (Forgy and others, 1984).

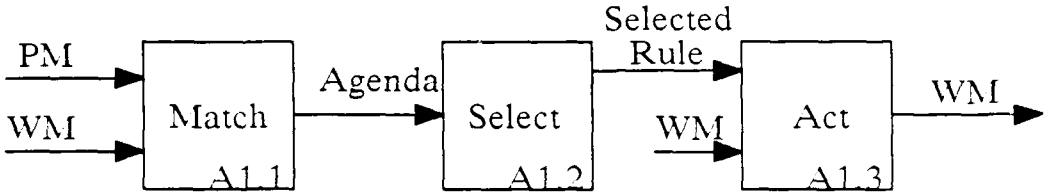


Figure 4-6. Control Flow of Inference Engine.

(Winston and Horn, 1985). This model of a production system had to be modified for the new data structure and functionality required by the RAV ART system.

The third phase was the design of the interface between ART and the newly designed inference engine. One design decision was to preserve the "integrity" of the design as much as possible. Therefore, a process was needed to translate the ART rules into a form usable by the new inference engine. The data flow diagram for this translation module can be seen in Figure 4-8. This interface allows for expanding this system with additional ART functionality with "minimal impact" on the inference engine. "Minimal impact" implies that the changes to the inference engine will be small and localized to several routines.

Parallel Inference Engine Design

The previous designed inference engine was modified slightly for parallel implementation. The algorithm for the parallel design is fashioned after the algorithms in chapter three and can be seen in Figure 4-9. The actual change in the design to the serial inference engine is small. The changes occur in the select and act phase. Each PE of the

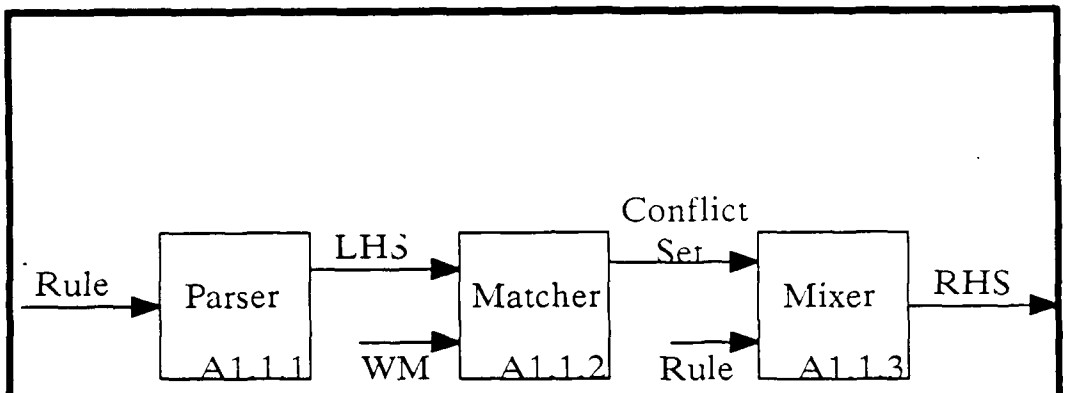


Figure 4-7. Expanded Control Flow of Inference Engine.

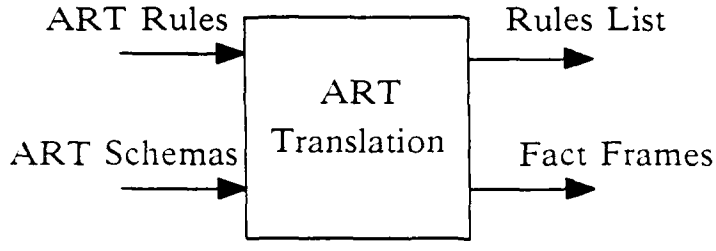


Figure 4-8. Data Flow for ART Translation.

1. Initialize: Place a copy of the simple inference engine on each PE. Place a copy of WM on each PE. Place a subset of the PM on each PE.

2. Repeat until done;

3. Match and select on each PE.

4. Report selection up tree.

5. Overall selection made at root node.

6. Broadcast WM change to all PEs.

7. end repeat;

Figure 4-9. Algorithm 8 - Parallel Inference Engine.

parallel design needs to report the rule it has selected for a system wide selection. The PE then has to wait to receive the system wide selection in order to act to update the working memory. The choices for the communication network included a star, a binary tree, and a spanning tree. The communication network for the flow of information is designed as a spanning tree. This can be seen in Figure 4-10 for a 16 node system. This reason for this type of tree is because it preserves nearest neighbor connections and the height of the tree is log p where p is the number of PEs. A node only communicates with other nodes a distance of one away and the length of a path from the bottom of the tree to the top is the dimension of the cube.

Basic parallel inference engines were designed in the last section. The only designing remaining is the methodology of placing rules of the RAV production system components on the parallel system's PEs. The first design to place the rules on the PEs, placed the rules by component on the different PEs. This was unsatisfactory since this

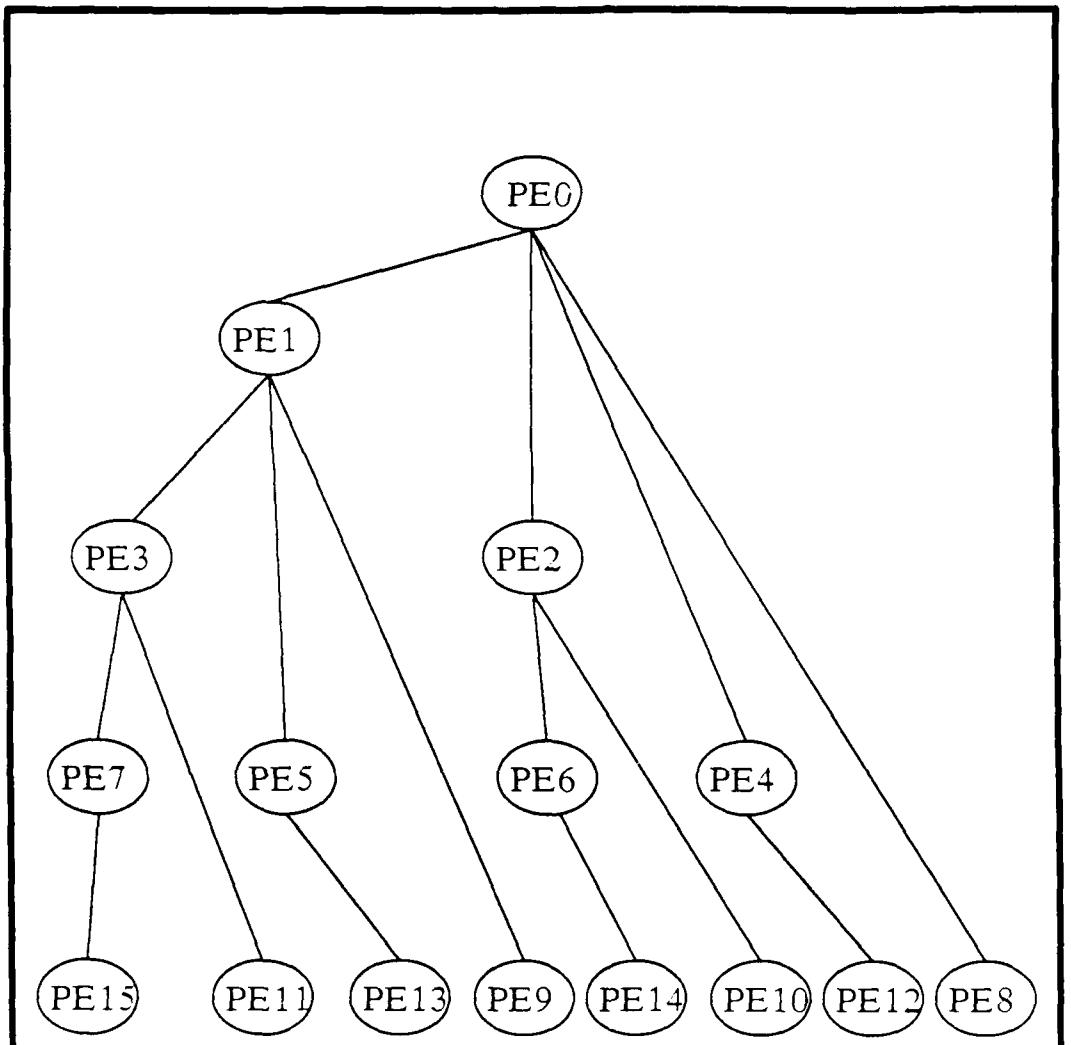


Figure 4-10. Spanning Tree Connections.
(Brandenburg and Scott, 1986).

produced an uneven load balance. Therefore, a more appropriate decomposition of the rules was to equally distribute the rules to the various PEs (Figure 4-11) .

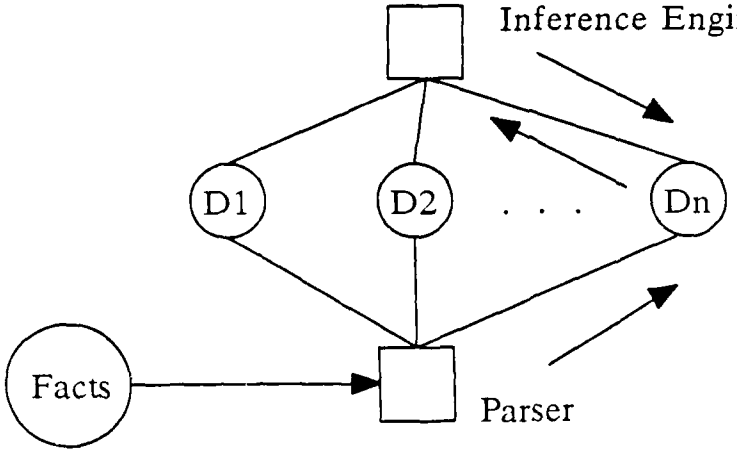


Figure 4-11. Rule Distribution.

Summary

This chapter provided the analysis of the RAV system and an inference engine design. Designs for a serial inference engine, a parallel inference engine, and two parallel RAV systems. The first design of the parallel RAV system did not take advantage of the independence of data elements of the components of the RAV system, while a second design tried to take advantage of this situation. All the designs were implemented and the results of that implementation are presented in chapter five.

V. Low-Level Design, Implementation, Experimental Results, and Analysis of Results

Introduction

This chapter provides the results of the implementations of the designs in chapter four. The basic measurements (performance metrics) of the RAV production system are defined and provided in this chapter. Various performance measurements are also provided for the RAV production system on both the TI Explorer LISP machine and the Intel iPSC hypercube. The "unique" code for the serial and parallel inference engine is provided in Appendix D. Salient features of the code as well as critical implementation criteria and problems are described in this chapter.

Inference Engine Low-Level Design

There were several choices for the initial inference engine. The first choice was a very basic system from Winston and Horn's book on LISP (Winston and Horn, 1984). The other choices were to implement algorithms from various papers (Miranker, 1987; Gupta, 1986). These other choices were high level designs of complex algorithms. The Winston and Horn (W&H) was in working code and was simple. This allowed for an incremental development of new capabilities. This code, however, was not suitable for the RAV rules and facts. This section will describe the original W&H system and then present the final expanded form of the inference engine on each of the three major sections of an inference engine: match, select, and act.

The first major section of an inference engine is the match routine. The syntax of the match pattern is shown in Figure 5-1. The original inference engine used all of these forms to match rules. Figure 5-1a is a simple match that requires the fact to be in that exact form. Figure 5-1b is a variable binding match where "plane" is bound to the value that corresponds to the first position in the fact. The rest of the fact is the same as the pattern. The pattern in Figure 5-1c uses an already existing variable binding to match

a. (F-16 airspeed 500)

b. ((> plane) airspeed 500)

c. ((< plane) airspeed 500)

d. (? airspeed 500)

e. (+ airspeed 500)

Figure 5-1. Original match patterns.

a fact. It uses the value of the binding to match against the first position of the fact. The rest of fact has to match the pattern exactly. Figures 5-1d&e are variable match patterns without binding. Figure 5-1d will match only those facts that only have one value before 'airspeed 500'. Figure 5-1e will match any fact whose last two positions are 'airspeed 500'.

The new inference engine's additional pattern match formats are show in Figure 5-2. Figure 5-2a shows the schema format. The word 'schema' is a key word. The

a. (schema name (slot value))

b. (schema name (slot (> value))

c. (schema (> name) (slot value))

d. (schema (> name) (slot (> value))

e. ((:> var) (test var))

f. ((- name))

g. ((>+ var))

Figure 5-2. New Match Patterns.

name has to match a defined schema name and the slot a valid slot within the schema name. The value in the pattern has to match the value of the schema slot. In Figure 5-2b the 'value' in the pattern is then matched and bound to the value of the schema 'name' and slot. Figure 5-2c and 5-2d are just variations on the variable binding theme with the schema and even the slot name being able to take on variable bindings. Figure 5-2e illustrates a test on a variable. The value of the fact is bound to the variable 'var' only if the test, which can be a function of 'var' is true. Figure 5-2f just provides for negation of a pattern. The Figure 5-2g was an optional part of the W&H implementation and provides a variable binding for a list of facts instead of an individual fact.

Figure 5-3 illustrates a sample rule. Line 1 of Figure 5-3 contains a rule name, an IF part, and a THEN part. The IF part consists of facts or facts with variables that could take on bindings. Line 1 of Figure 5-1 illustrates a simple fact that will only match a similar fact in the fact list. Line 2 of Figure 5-1 shows a simple variable binding. This clause will match any fact that begin with a single word and end with 'is a dog.' The variable name will be bound to the corresponding word in the fact. This binding can be used with the syntax in line 12 to pull the value of the variable to match elsewhere with the clause or rule. In the case of line 12, the binding of name is used to retract the fact (retraction was not an original part of the W&H system). Line 9 illustrate using the '+' symbol to match a group of symbols in a fact. Line 9 will match 'Joe is a dog,' 'Joe is dog,' 'Joe dog,' or 'Joe is a good dog.' The example on line 10 will not only match the same facts but will bind the phrase before 'dog' to the variable 'name-of.'

The changes in the match section can be divided into two types. The first is in more complex matching and binding strategies. The second is in a change or enhancement of the data structure of the facts. The match section had the most changes or enhancements and was probably the most difficult. The original W&H system placed all the facts into a list and matched all the rules against this list. As seen from the design, this proved too impractical for a system the size of the RAV system. Therefore, the rules from

```
(Rule Rule-name  (salience (sal-exp))
   (IF
1 -  (Joe is a dog)
2 -  ((> name) is a dog)
3 -  (schema airspeed (value 20))
4 -  (schema airspeed (value (> speed)))
5 -  (schema (>  plan) (value 20))
6 -  (schema (> plan) (value (> speed)))
7 -  (schema airspeed (value (:> speed (< speed 500))))
8 -   (binding speed 50)   )
9 -   (+ dog)
10 -  ((+ name-of) dog)

   (THEN (
11 -    (assert (Joe is a cat))
12 -    (retract ((< name) is a cat))
13 -    (modify airspeed (value 30))
14 -    (modify airspeed (bindings ((< bind) (hdg 30))))
    )) )
```

Figure 5-3. Rule Format.

the RAV system were already conveniently organized into schemas or frames, so the match and act function had to be changed to work with facts in the form of frames without losing the capability to keep a list of simple facts like the original W&H system. For antecedents in the form of line 3 or 4 of Figure 5-1, this was not too difficult. The routine that passed the clauses to be matched to the matcher was changed to look up the value of the slot and send the matcher that value along with the clause following the slot-name. This worked well. However, the system also needed to be able to handle a clause like line 5 or 6 of Figure 5-1. This was much harder. A list of all active schemas was introduced. The routine that passes clauses to the matcher has to look up in this list the name of a schema. With this name it looks up the slot value and passes this value along with the rule clause to the matcher. This routine does this for every schema in the list. This takes a great deal of time and is not practical. However, no better solution has been implemented. The clause form of line 7 in Figure 5-1 was also implemented. This allows a test of the value of a slot. Any test can be placed after the match variable.

The select phase of processing was the next section to require enhancements. The control of the selection of a rule to be selected had to be delayed. The W&H code only matched until a rule was found that completely matched, then it was selected and fired. This had to be changed so that all the rules would be looked at for a possible match before any rule would be selected. This meant that the eligible rules had to be kept on a list. Then when all the rules had been matched, this list would contain all the rules could "fire" or enter the act phase. This list of rules is called an "agenda". Once the agenda has been created then a selection of a rule is needed based on some criteria. This criteria has several alternatives. The first rule on the agenda could be selected. A random rule on the agenda could be selected. The rule with the most "specific" antecedent, the one with the most clauses could be selected. However, ART rules require that a "salience" be used to select the rule. This salience is a number associated with each rule determined by the author of the expert system to aid in the selection of rules. The highest salience is se-

lected first. Any ties among the salience's are then determined in a random fashion. This is a requirement stemming from the observation of the ART system. This change was instrumental to being able to implement a parallel inference engine which is be discussed in more detail later in this chapter.

The last phase that had to be changed was the act phase. This section changed for two reasons. First, addition functionality had to be incorporated due to the introduction of the schema system to implement facts. This required the need for the assertion, modification, and retraction of facts within schemas as well as the assertion and retraction of ordinary facts. The retraction function also required another change in the original W&H inference engine. This required that the act portion of the system check to see if a fact was already deleted or asserted so that the system would not go into a loop asserting or deleting the same fact. The original system did not allow for the modification of a fact. This is a function for schema facts only (ART 3.0, 1987). This allows the modification of a schema slot without first matching that slot. The slot can be directly changed. The second major change occurred due to the changes in the select portion of the system. The inference engine now had to maintain a list of matched rules. This was done in the act portion of the original W&H inference engine. Where the original system selected and acted upon a rule, this system matched and placed on the agenda a prospective rule. Then when all the rules had been matched, one rule was selected and fired.

The changes to the original W&H inference engine were rather extensive, but were incorporated without major impact. This provided a level of integrity to the original system which appeared to be sound. The code of the enhanced inference engine is contained in Appendix D.

Parallel Inference Engine Implementation

The parallel inference engine was implemented in CCLISP on the iPSC hypercube. Since the serial inference engine was implemented on the TI Explorer using Com-

mon Lisp the changes needed due to language differences were minimal. CCLISP (Brockhuysen, 1987b) was not as extensive as the Common Lisp on the TI Explorers (Explorer, 1985). For example, CCLISP did not support CADDDR, but this was easily changed. The language issues simply did not provide a major obstacle. There was, however, major effort involved in implementing a parallel inference engine. This centered around the communication between nodes.

The parallel design of the inference engine required that the selected rule from all the nodes be collected at one node for the final selection, and then that selection needs to be passed to all the other nodes. This can be done in several ways depending on the chosen communications pattern. Three alternatives seem appropriate: a star, a binary tree, or a spanning tree. With the star, one node acts as the central point with all other nodes communicating with that node. This would require longer than nearest neighbor communication or one node hops. The binary tree can be implemented with nearest neighbor communication, but only on higher dimension cubes. The spanning tree offers the appropriate functionality with nearest neighbor communication. An example of a spanning tree can be seen in Figure 5--4. A rule selected on node 14 would be sent to node 6. At node 6, this received rule would be added to the agenda and node 6 would select a rule. This continues up the tree until node 0 receives all the selected rules from its children. It then selects an overall rule and passes it down the tree to all its children. Each child then passes the selected rule to its children until all nodes receive the selected rule for firing. The algorithms for determining the parent and children nodes of the tree dependencies on a logical "or" of the binary node numbers. This was very hard to implement in LISP, so a table look-up was used. This proved to be very simply and efficient, but somewhat inflexible since only node zero can be used as the root node. The message passing within CCLISP presented some problems. There were several ways to pass messages. They ranged from low level message passing to high level FASL node streams. The low level message passing required that the length of the message length be known. This

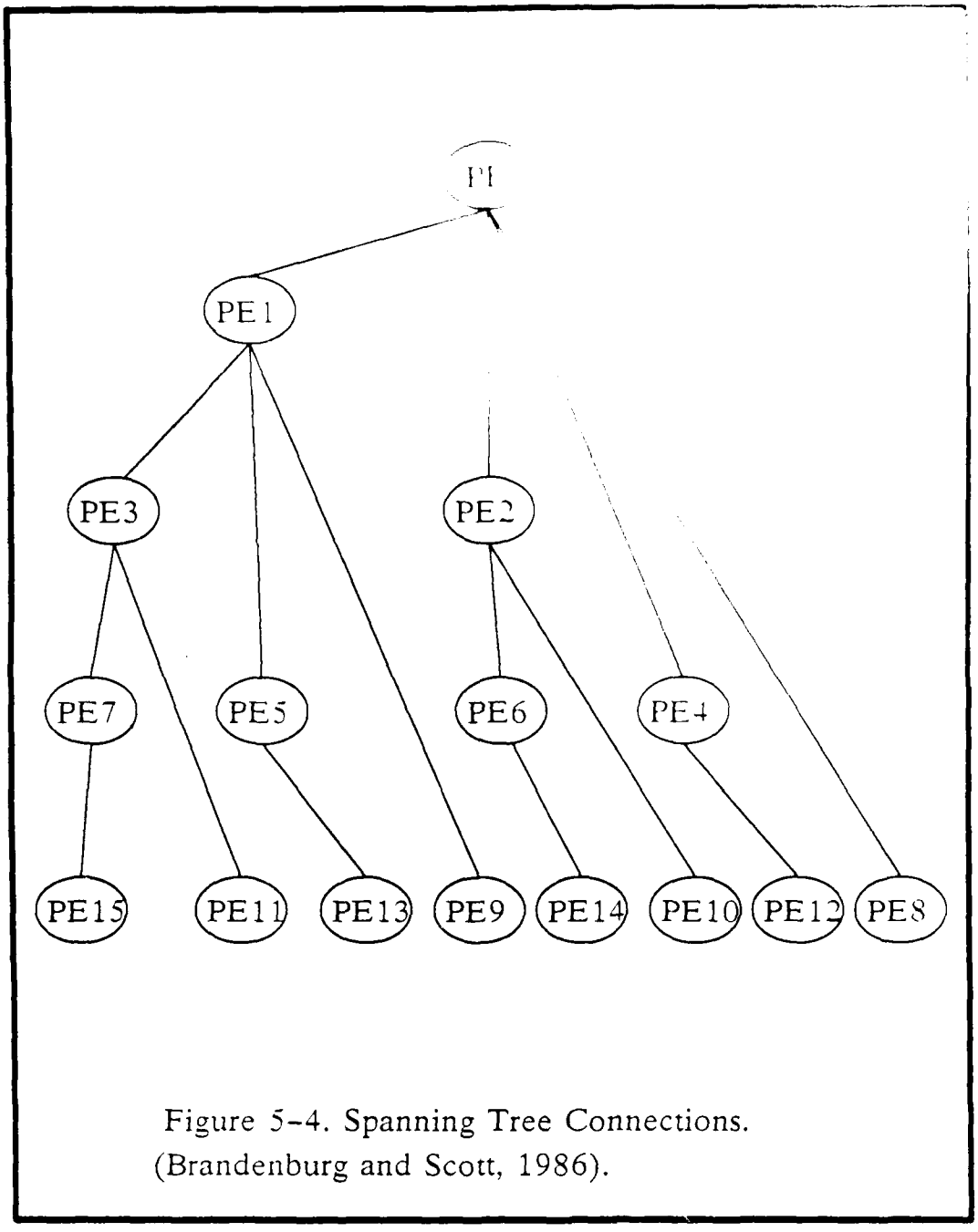


Figure 5-4. Spanning Tree Connections.
(Brandenburg and Scott, 1986).

proved to be a major limitation, given that the messages to be passed would be variable length rules. Therefore, the high level FASL node streams were selected for their abstraction. These streams did provide a problem. There was no defined way to do a receive-wait. This allows a process to enter receive mode until a message is received. This process is very convenient for synchronizing nodes. This function had to be built using a loop doing repeated receives until a message was received from another node. One other note concerning this process. The documented function 'listen' did not work (Broekhuysen, 1987b). This would have provided a method to test the message buffer for an incoming message without actually doing a read, but this function was not implemented.

The changes to the actual serial inference engine were small and confined to a small number of modules. These modules contained in Appendix D were "forward-chain" and "selected-rule." The first module had to be changed to provide the proper termination test. This is important to insure that the individual nodes terminated only when no overall rule was available, not just when the node found no matches. The other module had to be changed to incorporate the communications with the other nodes. Several other routines (found in Appendix D under the parallel code) were needed to assist this latter module to make the communication.

RAV System Implementation

The RAV system consisted of the original components of the RAV expert system designed by TI. In its original form it consisted of a series of plans, needs, and schemas which was a higher level abstraction than the ART rules (McNulty, 1987). The plans and needs were then "compiled" down into ART rules for execution using software developed by TI (Lystad, 1987). The only way to get the schemas and rules from the plans and needs was to compile the plans and needs into files rather than into the ART system. From there the rules and schemas are then transformed into a format that the serial and eventually the parallel inference engine could accept. This transformation was partially

automated with a routine (Appendix D) and further transformed by hand to come up with the final format compatible with the implemented inference engine. The total translation was not done programmatically due to the complexity of the software involved to parse and recognize the various ART syntax forms. This process could have been an entirely separate study.

The only test suite available was a demonstration developed by TI midway through the development of the system. In fact, the expert system used in this study was not complete and was only a demonstration prototype (Graham, 1987). This demonstration was considerably lengthy and required the perfect execution of all the rules and implementation of all the functionality of ART that was used with these rules. The alternative was to develop small prearranged sets of facts that would trigger a subset of rules. This was the preferable choice since the inference engine can not deal with all the rule format either in their entirety or efficiently. The complete demonstration was not used.

The code and expert system for the parallel RAV system was transported to the iPSC hypercube from the TI Explorer to a microVAX to a VAX across the Defense Data Network (DDN) to the AFIT VAX and finally to the iPSC hypercube (Fanning, 1987). This was perhaps the most "trying" of the problems associated with this whole implementation. This was because of the many machines that had to be traversed to get the code from the TI Explorer to the iPSC. This was only done after the tape-to-tape transfer failed due to a mismatch in tape-formats.

Testing Format

The test plan is divided into three components. The first is metrics dealing with the RAV expert system. The second is the metrics concerning the serial inference engine on the TI Explorer and the iPSC Hypercube. And, the third is the parallel inference engine on the iPSC Hypercube. This section describes the testing procedures and the results.

The measurements on the expert system were taken to provide a baseline foundation in order to compare this system with other expert systems. The items quantified for the rules were the number of rules, the number of clauses in the antecedents, and the number of clauses in the consequent. This part of the test calculated the number of schemas and from this calculated the number of facts represented in this manner. The results from this component of the test is shown in Figure 5-5.

For the last two components, similar tests were used. They consisted of using two sets of rules and three data sets. The two data sets were chosen due to the large run-time of the entire rule set. The most time consuming rules were contained in the mechanisms component. For this reason, one set of tests were conducted with this component and most all of the other tests without the mechanisms component. The data sets were chosen for the number of rules that they would cause to fire. This gives data over varying length of time and iterations through the match, select, act cycle. Figure 5-6 summarizes the two sets of rules with the three sets of data on the TI Explorer. The larger two sets of data were not run with the larger rule set due to the extremely long run times involved. Figure 5-7 through 5-9 summarize the results from the first data set that produced no rule firings with the first rule set. Figures 5-10 through 5-14 show the second data set that produced ten rule firings with the first rule set. And Figure 5-15 through 5-16 show the third data set that produced 27 rule firings with the first rule set. For the second and third data set, timings were not available for the 32-node system. This is explained in the analysis of results section.

Analysis of Results

This section analyzes the results of the implementation and the testing. This section looks at the correctness of the code and the timing of the tests. This section will also try to explain any anomalies and problems with implementation.

Subsystem	# Rules	# IF Clauses	# THEN clauses
PES			
Autopilot	2	3	4
Commnav	5	11	10
Departures	19	47	38
Hold-Arc	38	121	76
Intercepts	39	113	78
Intercepts-New	13	43	26
Landings	11	21	22
Mission	9	11	18
Recover	18	29	36
Takeoffs	18	46	36
Targets	3	12	6
Targets-All	2	8	4
Subtotal	**177**	**465**	**354**
VCES			
Airwork	21	39	42
Airwork-New	51	102	102
Autopilot	2	3	4
Elevation	3	7	6
Heading	21	67	42
Speedbrake	12	27	24
Targets	2	6	4
Throttle	19	48	38
Subtotal	**131**	**299**	**262**
Misc			
Mechanisms	73	237	148
Totals	**381**	**1001**	**764**

Figure 5-5. RAV Production System Characteristics.

Data Set 1 -> Produces 0 Firings Rule Set 1 -> 304 rules
Data Set 2 -> Produces 9 Firings Rule Set 2 -> 367 rules
Data Set 3 -> Produces 47 Firings

	Rule Set 1	Rule Set 2
Data Set 1	31.7771 sec	113.7404 sec
Data Set 2	325.1647 sec	———
Data Set 3	1788.4382 sec	———

Figure 5-6. TI Explorer Results.

The code is analyzed for its effectiveness and correctness. The code performs slowly compared to systems like ART which runs at between 2-30 rules per second (Gupta, 1986). On the Explorer, the inference engine plods along at about one cycle every 30 seconds or 2 rules per minute. This is with the smaller rule base. With the larger rule base, the system runs one cycle every 113 seconds. On the iPSC Hypercube, the serial system runs at one cycle in about 11 seconds with the smaller database and in about 79 seconds with the larger database. This brings about two concerns that need explanation. First, why does 60 extra rules slow the process down so much? The reason for this is in the format of these rules. They are rules that have a variable binding on the schema name. This means they must go through the list of schema names looking for a match. These rules can not take advantage of the indexing created by the frames. This is a process not handled well by the inference engine. Second, why does the TI Explorer go slower than the iPSC Hypercube? The surface appearance is that the Common Lisp on

Number of Nodes: 1
Number of Rules: 304 Rules/Node: 304
Number of Rules Fired: 0
Execution Time: 11.585 sec

Number of Nodes: 2
Number of Rules: 304 Rules/Node: 152
Number of Rules Fired: 0
Execution Time: (seconds): Speedup: 1.8

Nodes	0	1
Total	6.455	6.475
Match	6.355	6.085
Select	0.100	0.385
Act	0.000	0.005

Number of Nodes: 4
Number of Rules: 304 Rules/Node: 76
Number of Rules Fired: 0
Execution Time (seconds): Speedup: 3.0

Nodes	0	1	2	3
Total	3.780	3.820	3.825	3.875
Match	3.590	3.050	2.885	3.030
Select	0.175	0.770	0.935	0.840
Act	0.000	0.000	0.000	0.005

Figure 5-7. Data Set 1 for Rule Set 1a.

Number of Nodes: 8
Number of Rules: 304 Rules/Node: 38
Number of Rules Fired: 0
Execution Times (seconds): Speedup: 4.5

Nodes	0	1	2	3	4	5	6	7
Total	2.430	2.570	2.520	2.570	2.525	2.575	2.475	2.575
Match	1.655	1.425	1.460	1.575	1.700	1.630	1.465	1.505
Select	0.765	1.135	1.050	0.990	0.820	0.940	1.000	1.065
Act	0.000	0.000	0.000	0.000	0.000	0.005	0.000	0.000

Number of Nodes: 16
Number of Rules: 304 Rules/Node: 19
Number of Rules Fired: 0
Execution Times (seconds): Speedup: 5.1

Nodes	0	1	2	3	4	5	6	7
Total	1.995	2.265	2.270	2.320	2.170	2.220	2.270	2.320
Match	1.305	0.655	0.665	0.765	0.870	0.875	0.800	0.720
Select	0.680	1.610	1.600	1.550	1.295	1.335	1.470	1.600
Act	0.005	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Nodes	8	9	10	11	12	13	14	15
Total	2.125	2.125	2.175	2.175	2.125	2.175	2.170	2.225
Match	0.630	0.780	0.800	0.815	0.880	0.755	0.870	0.830
Select	1.490	1.340	1.370	1.355	1.240	1.415	1.295	1.390
Act	0.000	0.000	0.005	0.000	0.000	0.005	0.000	0.000

Figure 5-8. Data Set 1 for Rule Set 1b.

Number of Nodes: 32
Number of Rules: 304 Rules/Node: 9.5
Number of Rules Fired: 0
Execution Times (seconds): Speedup: 4.8

Node	0	1	2	3	4	5	6	7
Total	1.865	2.365	2.415	2.415	2.315	2.365	2.420	2.415
Match	0.755	0.355	0.360	0.480	0.385	0.510	0.415	0.370
Select	1.100	2.000	2.050	1.925	1.930	1.855	1.995	2.045
Act	0.005	0.005	0.000	0.005	0.000	0.000	0.000	0.000

Node	8	9	10	11	12	13	14	15
Total	2.270	2.320	2.320	2.320	2.270	2.320	2.370	2.320
Match	0.335	0.450	0.480	0.495	0.510	0.495	0.400	0.515
Select	1.930	1.865	1.835	1.820	1.755	1.820	1.965	1.800
Act	0.000	0.000	0.000	0.000	0.005	0.005	0.005	0.005

Node	16	17	18	19	20	21	22	23
Total	2.175	2.175	2.175	2.175	2.125	2.175	2.175	2.175
Match	0.500	0.305	0.315	0.290	0.485	0.370	0.385	0.355
Select	1.670	1.865	1.855	1.880	1.630	1.800	1.785	1.815
Act	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Node	24	25	26	27	28	29	30	31
Total	2.075	2.075	2.080	2.130	2.300	2.075	2.125	2.125
Match	0.340	0.375	0.365	0.320	0.415	0.350	0.355	0.315
Select	1.730	1.690	1.710	1.805	1.610	1.720	1.760	1.805
Act	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Figure 5-9. Data Set 1 for Rule Set 1c.

Number of Nodes: 1
Number of Rules: 304 Rules/Node: 304
Number of Rules Fired: 10
Execution Time: 189.800 sec.

Number of Nodes: 2
Number of Rules: 304 Rules/Node: 152
Number of Rules Fired: 10
Execution Time (seconds): Speedup: 1.6

Nodes	0	1
Total	121.860	121.870
Match	118.375	72.190
Select	2.825	49.045
Act	.610	.590

Number of Nodes: 4
Number of Rules: 304 Rules/Node:: 76
Number of Rules Fired: 10
Execution Time (seconds): Speedup: 2.0

Nodes	0	1	2	3
Total	93.420	93.500	93.490	93.515
Match	88.110	37.865	31.665	36.380
Select	4.690	55.025	61.210	56.525
Act	.570	.555	.565	.575

Figure 5-10. Data Set 2 for Rule Set 1a.

Number of Nodes:8
Number of Rules: 304 Rules/Node: 38
Number of Rules Fired:10
Execution Times (seconds): Speedup: 2.7

Nodes	0	1	2	3
Total	70.160	70.315	70.265	70.315
Match	19.760	19.820	16.195	19.350
Select	49.750	49.865	53.425	50.340
Act	00.615	00.605	00.605	00.605

Nodes	4	5	6	7
Total	70.225	70.325	70.275	70.320
Match	61.285	18.700	16.215	17.415
Select	08.305	50.980	53.425	52.255
Act	00.595	00.595	00.590	00.590

Figure 5-11. Data Set 2 for Rule Set 1b.

Nodes: 16
Number of Rules: 304 Rules/Node: 19
Number of Rules Fired: 10
Execution Times (seconds): Speedup: 2.8

Nodes	0	1	2	3
Total	65.960	66.265	66.215	66.265
Match	12.065	07.200	07.780	09.375
Select	53.265	58.420	57.805	56.265
Act	00.605	00.595	00.605	00.595
Nodes				
Total	66.170	66.220	66.215	66.270
Match	12.285	11.110	08.925	13.605
Select	53.240	54.455	56.660	52.030
Act	00.590	00.610	00.595	00.600
Nodes				
Total	66.705	66.175	66.125	66.225
Match	0.7430	11.555	08.685	10.300
Select	58.005	53.975	56.785	55.285
Act	00.595	00.595	00.595	00.600
Nodes				
Total	66.120	66.175	66.175	66.225
Match	48.380	08.510	07.580	08.845
Select	17.100	57.020	57.945	56.725
Act	00.595	00.595	00.600	00.595

Figure 5-12. Data Set 2 for Rule Set 1.

Number of Nodes: 1
Number of Rules: 304 Rules/Node: 304
Number of Rules Fired: 27
Execution Times: 571.385 seconds

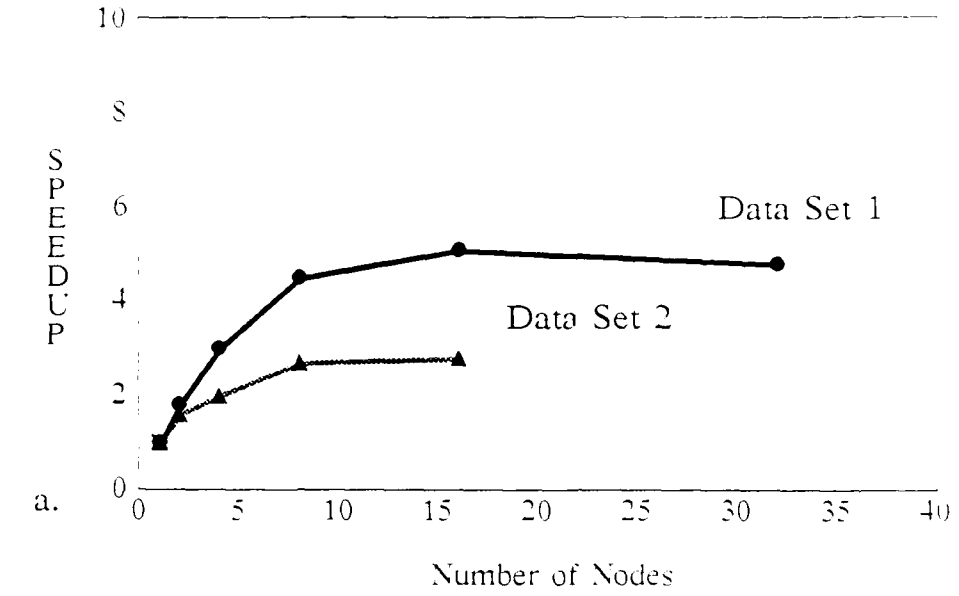
Number of Nodes: 2
Number of Rules: 304 Rules/Node: 152
Number of Rules Fired: 27
Execution Times (seconds): Speedup: 1.3

Nodes	0	1
Total	364.445	364.525
Match	354.670	206.840
Select	008.015	155.935
Act	001.655	001.665

Number of Nodes: 4
Number of Rules: 304 Rules/Node: 76
Number of Rules Fired: 27
Execution Times (seconds): Speedup: 2.1

Nodes	0	1	2	3
Total	278.430	278.470	278.475	278.475
Match	260.410	113.455	098.740	094.265
Select	016.250	163.305	178.015	182.465
Act	001.660	001.625	001.665	001.660

Figure 5-13. Data Set 3 for Test Set 1a.



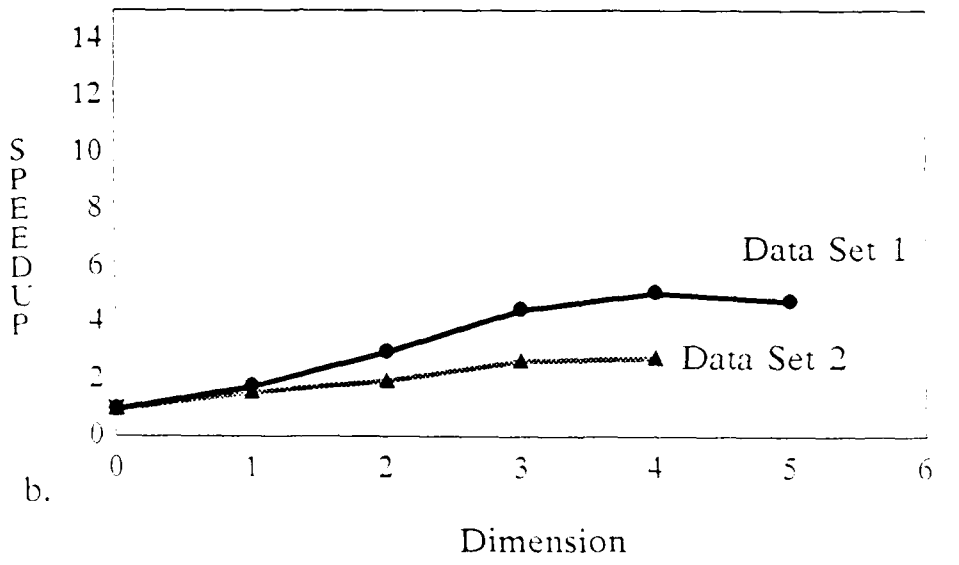


Figure 5-14. Speedup Graphs.

the Explorer is much more extensive than that on the iPSC Hypercube. These would create more overhead on the Explorer and allow the Hypercube to process faster.

This next section looks at the parallel performance of the inference engine. The speedups are far less than linear. The speedups taper off with the dimension of the cube. The answer to this problem can be seen in Figures 5-9 through 5-14. The select time on some of the nodes exceeds the match time on the node. In some instances, the select time far exceeds the match time. There are two explanations for this phenomena. The first is that the problem size is too small for the higher dimension cube. This can be seen from the first data set. On the higher dimension cube, the match time is less than the select time for one series of communication. The result is a much longer cycle than what would be expected from a linear speedup. The second cause for this slowdown is poor load balancing. Each node has an equal number of rules and each rule gets checked for a match on each cycle. The reason the load balance is off is due to the composition of the individual rules. The rules have different numbers of clauses that causes each rule to have a variable length match time. Also, the order of the clauses creates different match times. Even though a rule may have a long list of clauses to match, if the first one fails, then the rule matches quickly. Finally, the different types of matches take varying lengths of time depending on the format of the clause. A match involving a schema variable match will take much longer than a simple slot match.

VI. Conclusions and Recommendations

Introduction

This final chapter presents an analysis of results along with conclusions of the study. Application to other areas as well as recommendations for further study are also provided in this chapter.

Conclusions

This research study investigated the feasibility of parallel architectures to improve the performance of potential real-time software. In particular, the feasibility of parallel architectures to improve the NP-complete problem of state space search particularly in the form of a production system. The RAV expert system was used as an example of such a system for this study.

This study showed that poor speedups were obtained from the parallel implementation of a parallel inference engine using a relatively simple match routine. The speedups suffered from a combination of two factors. The first was a relatively small problem compared to the communications overhead. From chapter 5, it can be observed that for a system with greater than eight nodes, the time to perform the match cycle on a node was less than the time to communicate the selection. Also, the speedup suffered due to a poor load balance. Although the method for decomposing the rules seemed reasonable it proved to be totally unsatisfactory. The method did not take into account the variability among rules in the match process.

The performance of the iPSC hypercube to the TI Explorer came out fairly positively for the iPSC Hypercube. The iPSC Hypercube performed about twice as fast as the TI Explorer on the inference engine. However, this seems to be due to the simplicity of the LISP on the iPSC Hypercube.

The inference engine developed in this study performed adequately. The inference engine fired a rule about once every 30 seconds or at a rate of just under 2 a minute on the TI Explorer. The engine fired a rule one every 18 seconds on a single node of the iPSC Hypercube or just over 3 rules a minute. This does not compare with ART system that guage their systems in the rules per second, however, this was not a tremendous goal of the inference engine. The correctness of the inference engine was hard to analytically determine without further testing, but it appears that most of the ART functions used in the RAV expert systems were duplicated. The worst feature was matching a binding variable to the schema. This just would not work for a clause with a binding variable in the value for the slot. This part of the system would have to be redone to make a viable system.

The CCLISP on the iPSC hypercube is a mixture of good and bad. The good is that most of the TI Explorer code worked well. There were only a few minor glitches that only serve to emphasize the power of the TI Explorer LISP over CCLISP. The communication portion of CCLISP is still in its infant stage. The actual functionality was lacking and did not match the documentation (see Chapter 5).

The feasibility of transporting software from the TI Explorer to the iPSC hypercube, although troublesome on this attempt (see Chapter 5), seems to be a reasonable endeavor and with the aid of improved tools and techniques could become a routine process.

The speedup results from the inference engine were disappointing, however, it did show that speedups were possible. This study show the importance of load-balancing and placing a "large enough" problem on the iPSC hypercube. The timing is no where near real-time performance as was anticipated, however, with improvements in the inference engine and the load balancing significant improvements could be possible.

Applicability to Other Areas

This section looks at how this study might be scaled or applied to other areas. First a look at scaling the problem. From the results in chapter 5, it is apparent that the problem was not large enough to gain significant improvements. The RAV system contained over 300 rules which came out to just 10 rules per node on a 32 node system. Even with this crude inference engine, those rules could be processed in under 500 milliseconds. The communication on the tree took upwards of 1 to 2 seconds. Therefore, a scaled up problem is not only possible, but desirable. From the results in chapter 5, a system a factor of ten larger would not be a problem with the 32 node system. The production system could not afford to go much lower in size. There were performance drops as it were with the 32 node system in one of the cases of chapter 5.

This study shows the feasibility of concurrency with any state space search that follows the same type of match, select, act phase of a production system where the matching takes a high percentage of the processing time. The areas of resolution and branch-and-bound search techniques might benefit from this parallelization techniques. The problem needs to be decomposable in order to produce an equitable load balance. The problem also needs to be large enough so that the pieces process longer than the communication between nodes.

Recommendations for Further Study

This study probably raises more questions than it answers. Beginning with the serial inference engine. An area of study would be the performance of inference engines. The characteristics of inference engines and their performance would have been invaluable to this research. More work could be done to improve the inference engine in this study. The inference engine in this study could be redone using the Rete algorithm. The benchmarking of inference engines and inference engine techniques would be valuable. Also, with regard to inference engines, this study started to automate and simulate the

functionality of ART on the iPSC hypercube. The further development of the process could provide a valuable tool for expert system development. The expert system could be developed on the TI Explorer using ART and transferred to the iPSC hypercube to performance studies if a translation process were automatic.

The time complexity of the matching of various types of clauses within a rule varied. The inference engine was very slow matching some types of rules. These types of rules could benefit from a Rete-type match while the more simple rules could use a simple match. One area of possible future research would be to investigate the use of different types of inference engine on the different nodes of a parallel architecture to handle varying types of rules.

Only one parallel architecture was used in this study. How do other architectures compare with the architecture used in this study? The communication network in this study was a spanning tree. Is this the best choice? Would a binary tree, a star or some other pattern be better? The RAV software showed promise for further levels of concurrency. The current study used a complete set of facts on each node and only one rule was fired at a time across the entire network.

There needs to be better ways to characterize the work needed to match a rule so that more effective load balancing can be performed. This study ran out of time before effective ways to distribute the load could be worked out. This would depend on the structure of the inference engine, the structure of the rules and the structure of the facts. The load balancing can easily be "tweeked" by hand, but this should occur automatically.

The structure of the RAV expert system shows great promise in firing several rules in one cycle of the inference engine. This could provide a tremendous time and space savings. The RAV expert system is to operate in a real-time environment. This means a varying time requirement for operations. A parallel machine provides a predictable increase in processing power. The implementation of an automatic way to dynami-

cally increase the speed of a computation through a meta-level of knowledge and control would be very valuable to the design of real-time software to meet changing needs.

Appendix A: **Robotic Air Vehicle**

The Robotic Air Vehicle (RAV) is a concept under exploration by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Wright Aeronautical Laboratories (AFWAL). The concept is to create an unmanned air vehicle capable of autonomous operation. The RAV needs to be able to perform basic piloting skills as well as passive terrain following, terrain avoidance, obstacle avoidance, and autonomous navigation. The mission of such a vehicle would consist of intelligent reconnaissance or attack of high risk, heavily defended targets. A contract was awarded to Texas Instruments Incorporated (TI) in September 1985 to develop a system architecture as well as to demonstrate the feasibility of some of the key components of such a system (McNulty, 1987).

TI developed a system architecture (Figure A-1). The system centers around a piloting expert system. This module is responsible for the overall control of the RAV. This module directs the vehicle control expert system to perform basic maneuvers in which the vehicle control system could use to control the main interface to the RAV through the throttle, stick, and several switch controls. The piloting expert system also receives directions from a menu subsystem, a voice recognition subsystem, and a prepared mission plan. The piloting expert system could also receive information from an airspace expert system to provide a sense of situational awareness. Part of this situational awareness would come from a passive navigation subsystem that would provide the current vehicle location through passive sensors and a digital map. Projected future subsystems include a mission replanner and a threat assessor, but these are not currently being worked on by TI (Blair, 1986; Graham, 1987; McNulty, 1987).

As indicated by some of the names of the subsystems, the method of implementation was chosen to be artificial intelligence expert systems using production systems. This method was chosen by AFWAL since other methods to control a vehicle failed (Blair, 1987). The code for such systems become large and unmanageable without provid-

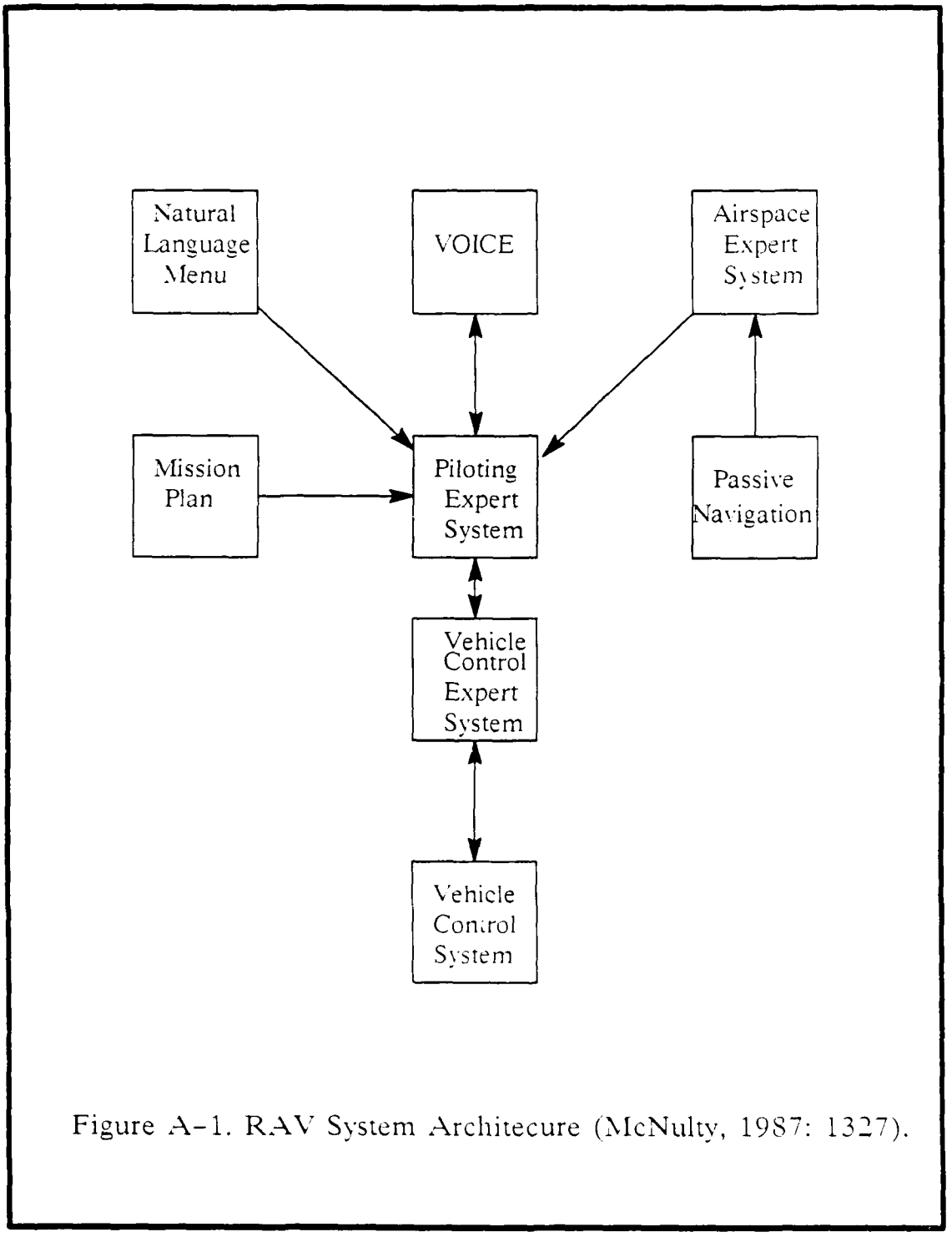


Figure A-1. RAV System Architecure (McNulty, 1987: 1327).

ing a solution. With the exception of the voice control and passive navigation subsystems, all the other subsystems were implemented as some type of expert system (McNulty, 1987).

The design of the expert systems was very simplistic in nature. The piloting skills were designed and developed using the pilot training paradigm. The expert system was built up and tested using the same basic skills and in the same order as a human pilot learns them in pilot training. The system was designed using the Automated Reasoning Tool (ART). ART allows the specification of rules and facts that can take the form of schemas or relationships. This level of abstraction was a bit too low for easy design. TI, therefore, abstracted the design one level higher to plans and needs. This made thinking about the pilot skills easier without getting to the rule level. If a high level action was needed a need was built. The need would then get accomplished by a plan. The needed rules and data structures would then be built to track the need and plan. Meta-level rules were developed to handle the activation and deactivation of plans and needs. Most of the facts in working memory were designed as frames within ART known as schemas. This allows for fast access of the facts and also provide an effect way to organize the facts. This was particularly useful in the parallel design.

The system used for this study was demonstrated in May 1987 with the following status of the subsystems. The piloting expert system (PES), the vehicle control expert system (VCES), the vehicle control system (VCS) and the airspace expert system (AES) all have operative basic functions for a limited scope. The passive navigation has basic theoretical problems. The voice recognition is somewhat limited in its capability due to the state-of-the-art in this area. The menu system and mission planner are at a very basic and simplistic state (Graham, 1987). It is due to this status that the scope of the thesis is limited to the four major components mentioned above: PES, VCES, VCS, and the AES. The AES has been studied in less detail since its major component is a relational database. It does contain an expert system and could benefit from this study.

It should be noted that the three modules that are used to control the vehicle (PES, VCES, VCS) conform to a theory by Saridis on intelligent robotic control. It is a hierarchically intelligent control approach proposed to unify cognitive and control systems theory. It uses the principle of decreasing precision with increasing intelligence. Saridis proposes a three layer approach. The top layer is the organization layer which controls and supervises the overall activity. The middle layer is the coordination layer. This level controls the subtasks to be performed as ordered by the organization layer. The bottom layer is the hardware control level. This level controls the basic functions and movement using mathematical models of motion (Saridis, 1983). Figure A-2 shows the correspondence with this layered approach and the RAV control.

1. Organization Level – Piloting Expert System (PES)
2. Coordination Level – Vehicle Control Expert System (VCES)
3. Hardware Control Level – Vehicle Control System (VCS)

Figure A-2. RAV Intelligent Control Layers (McNulty, 1987; Saridis, 1983).

APPENDIX B: **Parallel Processing Architectures**

Computer architectures can be divided into four categories. These categories are Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD). The SISD represents all the single processor systems. The SIMD, MISD, and MIMD represent the parallel processor systems. The MISD architecture has been shown to be infeasible for pratical applications since there is little use for concurrent multiple operations on a single data stream. This leaves the SIMD and the MIMD as the two main categories of parallel computer architectures. The SIMD is a system with multiple processors operating the same instruction synchronously on separate data streams. Examples of this type of architecture is the Cray and Connection machines. The MIMD architectures have multiple processors capable of operating on multiple data streams with different operations asynchronously. Examples of this type of architecture are the Butterfly and iPSC hypercube (Hwang and Briggs, 1984).

The SIMD and MIMD categories are further broken down into subclasses. These classes are based on the connection network that allows the separate processors to communicate with each other. This is important since it is often impractical for all the processors to be connected to each other. Several types of networks are a line, a ring, a star, and an n-cube (Hwang and Briggs, 1984). These can be seen in Figure B-1.

There are several other characteristics of parallel architectures. The first is the type of memory organization. Some systems have only a common main memory, or only memory local to each processor, or a combination of both. The number and power of each processor is also an important characteristic of the system.

Several architectures were considered for this study due to availability and applicability to the problem. The three architectures considered were a network of TI Explorer Lisp machines, an Intel iPSC hypercube, and a Connection machine. Each of these

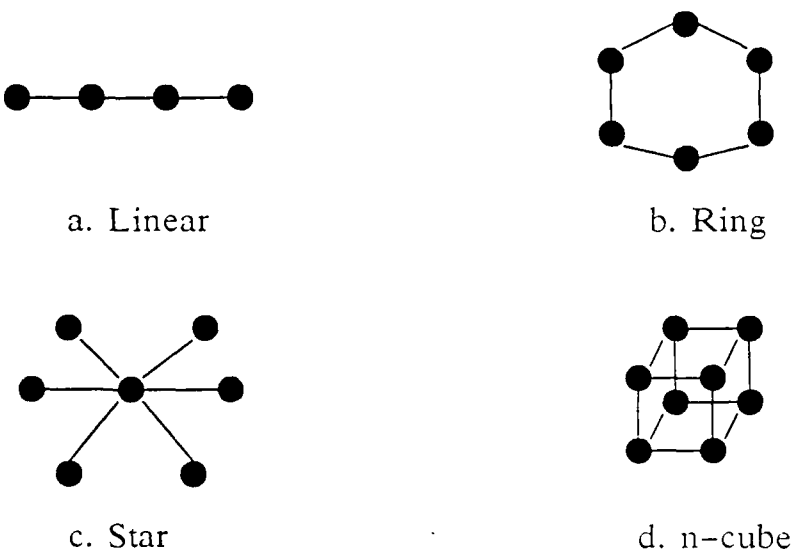


Figure B-1. Interconnection Networks.

machines will be discussed in detail.

The first architecture is the host architecture for the RAV system. It is a network of four TI Explorer Lisp machines. They form a loosely connected system with each system having its own local memory with a common bus structure between the systems. They all share a central file server. Each of the four systems are very powerful Lisp processors. This architecture was chosen for this study.

The second architecture is the Intel iPSC hypercube. This system consists of up to 32 processing elements (PE) with memory expansion. This gives each PE four megabytes of memory. The connection network for the system is an n-cube. This means that the worst case communication length is log n processors. Each processor consist of an Intel 80286. The advantage of an n-cube interconnection network is its ability to simulate many other connection networks including a tree. This flexibilty along with ready accessibility made this architecture a very natural choice for this study.

The third architecture considered was the Connection Machine. This architecture is of the SIMD flavor. It consists of up to 64 K one bit processors with 1 K of local memory (Hillis, 1987). Although this architecture proved interesting it was not chosen for this study for several reasons not specifically related to its architecture. The major constraints were time, easy access, and inexperience with the architecture and methodologies.

The previously discussed architecture are summarized in Figure B-2. Although other architectures and machines exist no other systems were evaluated due to availability.

SIMD processor arrays:	Connection Machine IBM GF-11 FPS 164/MAX ICL/DAP Loral MPP
MIMD Shared Memory:	Cray X-MP/2,4 Cray 2 Alliant FX/8 Encore/Multimax Elxsi 6400 Sequent 8000 Cray 3 IBM 3090/400 VF Univax 1194/ISP
MIMD Distributed Memory:	iPSC Ametek 14 NCUBE BBN Butterfly CDC Cyberplus Culler PSC FPS T-Series Warp

Figure B-2. Summary of Architectures. (Hwang, 1987: 1350).

Appendix C: NP-Completeness

NP-Complete problems are a class of computationally hard problems. These problems can be solved in polynomial time on a non-deterministic Turing machine. There is no known solution to these problems on a deterministic automata in less than exponential time. Exponential time means that in the worst case the time complexity of the problem has a lower bound that is an exponential function of the size of the problem. To show that a problem is an NP-complete, two properties have to be shown. First, the problem has to be shown to be combinatoric or have an exponential time complexity. Second, a known NP-complete problem has to be able to be transformed into the problem to be proved in polynomial time. A list of known NP-Complete problems can be seen in Figure C-1. The second condition of NP-Completeness insures one of the important characteristics of this class of problems. If any of the problems could be shown to exhibit less than an exponential time complexity, then all the problems could be transformed and solved in the same manner (Aho and others, 1974).

In this appendix, several problems are shown to be NP-complete. The RAV consists of one primary problem. This is the intelligent control of a robot. Also shown to be NP-complete is the generic solution method used for the RAV, namely production systems. This is added since this is a general methodology that occurs frequently in AI and perhaps the results can be more generally applied.

The intelligent control of a robot is shown to be NP-complete in a rather straight forward way. First, a precise definition of the problem is necessary. This problem can be described as a function H that take the input vectors C and F and produces an output P. In this case, C is a command vector and F is a feedback vector. P is the output commands or adjustments (Albus, 1981). There are (C+F)! combinations of possible situations that could occur. Clearly in the worst case all these combinations would need to be considered to actuate the proper control. This function obviously has a lower time

1. Satisfiability – Is a Boolean expression satisfiable?

2. Clique – Does an undirected graph have a clique of size k?

3. Vertex cover – Does an undirected graph have a vertex cover of size k?

4. Hamilton circuit – Does an undirected graph have a Hamilton circuit?

5. Colorability – Is an undirected graph k colorable?

6. Feedback vertex set – Does a directed graph have a feedback vertex set with k members?

7. Feedback edge set – Does a directed graph have a feedback edge set with k members?

8. Directed Hamiltion circuit – Does a directed graph have a directed Hamilton circuit?

9. Set Cover – Given a family of sets S1, S2, . . ., Sn does there exist a subfamily of k sets such that the union of the subfamily equals the union of the entire family?

10. Exact Cover – Given a family of sets S1, S2, . . . Sn does there exist a set cover consisting of a subfamily of pairwise disjoint sets?

Figure C–1. Known NP–Complete Problems. (Aho and others, 1974: 378).

complexity bound that is exponential. Next a known NP-complete problem has to be transformed into the intelligent control problem in polynomial time. The known problem selected is the Set Covering Problem. A definition of the Set Covering Problem is needed for this transformation. The Set Covering Problem is defined as: Given a family of sets S1, S2, ... Sn does there exist a subfamily of k sets such that the union of the subfamily equals the union of the entire family (Aho and others, 1974). The transformation would be as follows:

> 1. The sets to be included would be the C vector. If the set were included, then the component of the C vector would be 1, otherwise it would be 0.
> 2. The F vector would be the union of all the sets with a 1 in the C vector.

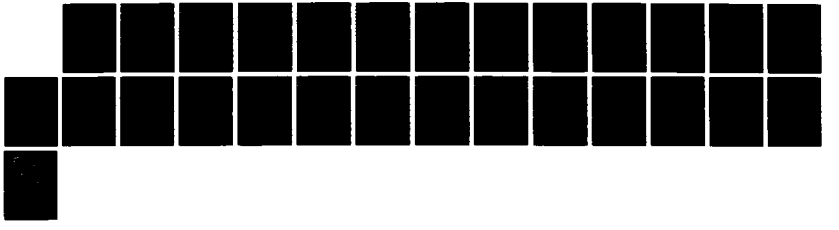
The transformation can obviously be accomplished in polynomial time. Therefore, the intelligent control problem is an NP-complete problem.

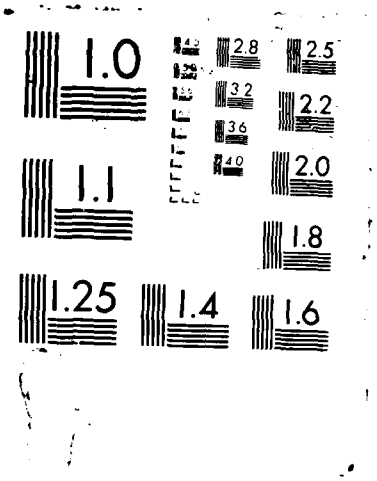
Before the intelligent control problem was properly defined, efforts to prove the problems associated with the RAV lead to the same effort of showing that an expert system or production system is NP-complete. Although a production system is not itself a problem, it is the solution method to many problems. Also due to its widespread use in AI, a proof of its NP-completeness is appropriate.

A production system is composed of productions or rules of the If-Then or antecedent-consequent form and a set of initial facts or axioms. The system can then produce new facts by applying the rules to the initial facts or any new facts previously generated. This type of a system was proven by Post to be able to compute any Turing computable function. Therefore any NP-complete problem can be transformed into a production system. The production system has a flaw that NP-complete problems do not, that is it is susceptible to the Halting problem. A production system is not guaranteed to halt. Due to its computational characteristics, production systems are [illegible]

AD-A190 603 PARALLEL ARTIFICIAL INTELLIGENCE SEARCH TECHNIQUES FOR REAL TIME APPLICATIONS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. D J SHAKLEY 2/2

UNCLASSIFIED DEC 87 AFIT/GCS/ENG/87D-24 F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

der predicate logic. This is not surprising since predicate logic has a production system form. Predicate logic has an initial set of axioms and a set of rules to produce new theorems. This is the formula for a production system.

Appendix D: **Code**

General Match Code

```
;;;*****************************************************************************
;;;*****************************************************************************
;;;**
;;;**
;;;**    TITLE: Inference Engine.
;;;**    DATE: 7 Dec 1987.
;;;**    VERSION: 1.0.
;;;**    FUNCTION: To perform the match, select, act cycle of a forward
;;;**                chaining inference engine.
;;;**    LANGUAGE: Common LISP.
;;;**    INPUTS: None.
;;;**    OUTPUTS: None.
;;;**    FILES READ: None.
;;;**    FILES WRITTEN: None.
;;;**    AUTHOR: Donald J. Shakley.
;;;**    HISTORY: Written for Master's Thesis. Portions taken from Winston &
;;;**              Horn's book on LISP.
;;;**     INSTALLATION: TI Explorer.
;;;**                   iPSC Hypercube.
;;;**
;;;**
;;;*****************************************************************************
;;;*****************************************************************************


(defvar assertions) ; variable for list of assertions or plain facts
(defvar rules)      ; variable for list rules
(defvar agenda)     ; variable for rules on the agenda
(defvar schemas)    ; variable for list of schema names

(Defun match (p d assignments)
  (cond ((equal assignments t) (setq assignments nil)))
  (cond ((and (null p) (null d))
         (cond ((null assignments) t) ; return true if no bindings
               (T assignments)))       ; return bindings

          ((and (atom p) (atom d)) ; test for atom equality
           (cond ((equal p d) (match nil nil assignments))
                 (t nil)))
          ((or (null p) (null d)) nil)

         ((or (equal (car p) '?) ; test for generic general item matcher
              (equal (car p) (car d)))
          (match (cdr p) (cdr d) assignments))

         ((equal (car p) '+)     ; test for generic general list matcher
             (or (match (cdr p) (cdr d) assignments)
                 (match p (cdr d) assignments)))
```

```
((equal (car p) 'S+)  ; test for general binding list match
 (cond ((equal d '(empty)) nil)
       (t
          (let ((find (assoc (caadr p) d)))
            (match (cadr p) find assignments)))))

((atom (car p)) nil) ; check for ill-formed pattern

((equal (pattern-indicator (car p)) '-) ; match negated item
 (cond
   ((not (equal (pattern-variable (car p)) (car d)))
    (match (cdr p) (cdr d) assignments))))

((equal (pattern-indicator (car p)) ':>) ; binding conditional match
   (cond ((test-condition (car p) (car d) assignments)
            (match (cdr p) (cdr d)
                     (shove-gr (pattern-variable (car p))
                               (car d)
                               assignments)))
           (t nil)))

 ((equal (pattern-indicator (car p)) '>) ; binding variable match
(match (cdr p) (cdr d)
           (shove-gr (pattern-variable (car p))
                     (car d)
                     assignments)))

((equal (pattern-indicator (car p)) '<) ; use binding variable match
 (match (cons (pull-value (pattern-variable (car p)) assignments)
              (cdr p))
          d
          assignments))

((equal (pattern-indicator (car p)) '+) ; binding list match
 (let ((new-assignments (shove-pl (pattern-variable (car p))
                                                  (car d)
                                                  assignments)))
   (or (match (cdr p) (cdr d) new-assignments)
       (match p (cdr d) new-assignments))))

((equal (pattern-indicator (car p)) '<+) ; use binding list match
 (match (append (pull-value (pattern-variable (car p))
                                          assignments)
                    (cdr p))
          d
          assignments))

((and (equal (pattern-indicator (car p))  ; restriction match
               'restrict)
      (equal (restriction-indicator (car p)) '?)
      (test (restriction-predicates (car p)) (car d)))
 (match (cdr p) (cdr d) assignments))
```

```
          (t ; none of above condition hold - dig deeper into structure
           (append (match (car p) (car d) assignments)
                   (match (cdr p) (car d) assignments)))
))

;;; Function to return the first item of a pattern

(defun restriction-indicator (pattern-item) (cadr pattern-item))

;;; Function to return second item of a pattern

(defun restriction-predicates (pattern-item) (cddr pattern-item))

;;; Function to place a variable/ binding on an association list

(defun shove-pl (variable item a-list)
  (cond  ((null a-list) (list (list variable (list item))))
         ((equal variable (caar a-list))
          (cons (list variable (append (cadar a-list) (list item)))
                (cdr a-list)))
         (t (cons (car a-list)
                  (shove-pl variable item (cdr a-list))))))

;;; function to pull a value of a variable off the association list

(defun pull-value (variable a-list)
  (cadr (assoc variable a-list)))

;;; function to place a variable/binding on an  association list

(defun shove-gr (variable item a-list)
  (append a-list (list (list variable item))))

;;; function to return the first value on a list

(defun pattern-indicator (l)
  (car l))

;;; function to return second value from a list

(defun pattern-variable (l)
  (cadr l))

;;;  function used to evaluate condition of a match

(defun test-condition (condition value bindings)
  (let ((clause (third condition))
        (obj (second condition)))
    (cond ((null value) (setq value 0)))
    (setq clause (subst value obj clause))
    (do  ((bind bindings (cdr bind)))
         ((null bind) t)
```

```
    (setq clause (subst (second (car bind)) (first (car bind)) clause)))
  (eval clause)))

;;; function to calculate a value of a binding

(defun calculate (clause bindings)
  (do ((bind bindings (cdr bind)))
      ((null bind) t)
    (setq clause (subst (second (car bind)) (first (car bind)) clause)))
(eval clause))

;;; function used with the restrict pattern match

(defun test (predicates argument)
  (cond ((null predicates) t)
        ((funcall (car predicates) argument)
         (test (cdr predicates) argument))
        (t nil)))

;;; function to retract a fact from the list of assertions

(defun retract (fact)
  (cond ((already-fact fact)
         (setq assertions (remove fact assertions :test 'equal)))))

;;; function to add a single fact to the list of assertions

(defun add-fact (fact)
  (cond ((not (already-fact fact))
         (setq assertions (cons fact assertions)))))

;;; function to add a list of facts to the list of assertions

(defun add-facts (facts)
  (setq assertions (append facts assertions)))

;;; function to modify a schema's slot value

(defun change (schema)
  (setf (get (second schema) (first (third schema)))
                             (second (third schema))))

;;; function to determine if a fact is already in a schema

(defun already-schema-fact (schema)
  (cond ((equal (second (third schema))
                (get (second schema) (first (third schema))))))

;;; function to place a selected rule actions into the database

(defun remember (new)
  (print '(******** rule ,(caar new) fired  **********))
  (setq new (cdr new))
```

```
  (do ((clauses new (cdr clauses)))
      ((null clauses) new)
    (cond
      ((equal (caar clauses) 'assert) (add-fact (cadar clauses)))
      ((equal (caar clauses) 'retract) (retract (cadar clauses)))
      ((equal (caar clauses) 'modify) (change (cadar clauses))))))

;;; function to determine if a fact is already in the list of assertions

(defun already-fact (new)
  (cond ((member new assertions :test 'equal) t)
        (t nil)))

;;; function to place a rule onto the list of assertions

(defun remember-agenda (new ruleused)
  (setq already-there t)
  (do ((clauses new (cdr clauses)))
      ((null clauses) (not already-there))
    (cond
      ((member (cons ruleused new) agenda :test 'equal)
       (setq already-there t))
      ((equal (caar clauses) 'assert)
       (setq already-there
             (and already-there (already-fact (cadar clauses)))))
      ((equal (caar clauses) 'retract)
       (setq already-there
             (and already-there (not (already-fact (cadar clauses))))))
      ((equal (caar clauses) 'modify)
       (setq already-there
             (and already-there (already-schema-fact (cadar clauses)))))))
  (cond
    (already-there nil)
    ((not already-there) (setq agenda (cons (cons ruleused new) agenda)))))

;;; function to combine two lists

(defun combine-streams (s1 s2) (append s1 s2))

;; function to place an item on a list

(defun add-to-stream (e s) (cons e s))

;;; function that returns the first of a list

(defun first-of-stream (s) (car s))

;;;function that returns the rest of a list

(defun rest-of-stream (s) (cdr s))

;;;  function to determine if a list is empty
```

```
(defun empty-stream-p (s) (null s))

;;; function to make an empty list

(defun make-empty-stream () nil)

;;; feeds clauses of a rule to the match function for matching

(defun filter-assertions (pattern initial-a-list)
  (cond
    ((and (equal (car pattern) 'schema)
          (listp (second pattern)))
     (do ((schemas schemas (cdr schemas))
          (a-list-stream (make-empty-stream)))
         ((null schemas) a-list-stream)
       (let ((new-a-list (match (list (second (third pattern)))
                                (list (get (car schemas) (first (third pattern))))
                                (list (list (second (second pattern)) (car schemas))))))
         (cond (new-a-list (setq a-list-stream
                                 (add-to-stream new-a-list a-list-stream)))))))

    ;;; simply schema match
    ((equal (car pattern) 'schema)
     (let ((new-a-list (match (list (second (third pattern)))
                              (list (get (second pattern) (first (third pattern))))
                                    initial-a-list))
           (a-list-stream (make-empty-stream)))
       (cond (new-a-list (setq a-list-stream
                               (add-to-stream new-a-list (make-empty-stream)))))))

    ;;; set up bindings
    ((equal (car pattern) 'bind)
     (list (shove-gr (second pattern)
                     (calculate (third pattern) initial-a-list)
                     initial-a-list)))

     ;;; regular fact match
    (t
    (do ((assertions assertions (cdr assertions))
         (a-list-stream (make-empty-stream)))
        ((null assertions) a-list-stream)
      (let ((new-a-list (match pattern (car assertions) initial-a-list)))
        (cond (new-a-list (setq a-list-stream
                                (add-to-stream new-a-list a-list-stream)))))))))

;;; sends filter-assertions different variable bindings of variables

(defun filter-a-list-stream (pattern a-list-stream)
 (cond ((empty-stream-p a-list-stream) (make-empty-stream))
       (T (combine-streams
            (filter-assertions pattern (first-of-stream a-list-stream))
            (filter-a-list-stream pattern (rest-of-stream a-list-stream))))))
```

```
;;; function to pass filter-a-list-stream the clauses of a rule one at a time

(defun cascade-through-patterns (patterns a-list-stream)
 (cond ((null patterns) a-list-stream)
       (T (filter-a-list-stream (car patterns)
                                (cascade-through-patterns (cdr patterns)
                                                          a-list-stream)))))

(defun cadddr (x) (cadr (cddr x)))

;;;  function to determine if a rule matches and apply the rule

(defun use-rule (rule)
  (let* ((rule-name (cadr rule))
         (ifs (reverse (cdr (caddr rule))))
         (thens (cdr (cadddr rule)))
         (a-list-stream (cascade-through-patterns
                          ifs
                          (add-to-stream nil (make-empty-stream))))
         (action-stream (feed-to-actions rule-name thens a-list-stream)))
    (not (empty-stream-p action-stream))))

;;; function to replace all the variables in the consequent and adds the rule to the
;;; agenda

(defun spread-through-actions (rule-name actions a-list)
  (do ((actions actions (cdr actions))
       (action-stream (make-empty-stream)))
      ((null actions) action-stream)
    (let ((action (replace-variables (car actions) a-list)))
      (cond ((remember-agenda action rule-name)
             (setq action-stream (add-to-stream action action-stream)))))))

;;; function to replace the variables with the value of their bindings

(defun replace-variables (s a-list)
  (cond ((atom s) s)
        ((equal (car s) '<)
         (cadr (assoc (pattern-variable s) a-list)))
        ((equal (car s) '<+)
         (cadr (assoc (pattern-variable s) a-list)))
        (t (cons (replace-variables (car s) a-list)
                 (replace-variables (cdr s) a-list)))))

;;; function to feed spread through actions all the various bindings of a rule

(defun feed-to-actions (rule-name actions a-list-stream)
  (cond ((empty-stream-p a-list-stream) (make-empty-stream))
        (t (combine-streams
             (spread-through-actions rule-name
                                     actions
                                     (first-of-stream a-list-stream))
```

```
                (feed-to-actions rule-name
                                 actions
                                 (rest-of-stream a-list-stream))))))

;;; variables to keep track of time and number of rules

(setq begin-time 0)
(setq end-time 0)
(setq total-time 0)
(setq number-of-rules 0)

;;; function to initialize the above variable every time inference engine is started

(defun initialize-main ()
(setq begin-time 0)
(setq end-time 0)
(setq total-time 0)
(setq number-of-rules 0))

;;; main body of inference engine

(defun forward-chain ()
(initialize-main)
(setq begin-time (sys:clock))
  (do ((done T))
      ((null done) (not done))
    (setq agenda nil)
    (do ((rules-to-try rules (cdr rules-to-try))
           (rules-tried 0))
          ((null rules-to-try) t)
      (cond ((use-rule (car rules-to-try))
               (setq rules-tried (+ rules-tried 1)))
               ((> rules-tried 25) (setq rules-tried 0)
                     (print '(ten more rules tried ,(length rules-to-try) left)))
               (t (setq rules-tried (+ rules-tried 1))) ))
    (cond ((null agenda) (setq done nil))
          (t (remember (select-a-rule agenda))
             (setq number-of-rules (+ number-of-rules 1)))) )
(setq end-time (sys:clock))
(setq total-time (- end-time begin-time))
(print '(total rules fired ,number-of-rules))
(print '(total time ,total-time)) )

;;; function to select a rule from the agenda

(defun select-a-rule (agenda-list)
  (do ((rules-to-select agenda-list (cdr rules-to-select))
       (selected-rule (car agenda-list)))
      ((null rules-to-select) selected-rule)
;;    (print '(rule ,(caar (car rules-to-select)) has salience
                    ,(salience (car rules-to-select))))
    (cond ((< (salience selected-rule) (salience (car rules-to-select)))
              (setq selected-rule (car rules-to-select))))))
```

```
;;; calculates the salience of a rule

(defun salience (rule)
 (let ((sal (cadr (cadar rule))))
   (cond ((numberp sal) sal)
         ((atom sal) (eval sal))
         (t (funcall (first sal)
                     (eval (eval (second sal)))
                     (eval (eval (third sal))))))))

;;; sets initial values of some important variables

(setq *maximum-salience*  1000000)

(setq *minimum-salience* -1000000)

(setq *default-salience* 0)

(setq agenda nil)

;*****************************************************************************
;*****************************************************************************
;**
;**
;**    TITLE: Inference Engine Utilities.
;**    DATE: 7 Dec 1987.
;**    VERSION 1.0.
;**    FUNCTION: To provide utility routines for the Inference engine.
;**    LANGUAGE: Common LISP.
;**    INPUTS: None.
;**    OUTPUTS: None.
;**    FILES READ: None.
;**    FILES WRITTEN: None.
;**    AUTHOR: Donald J. Shakley.
;**    HISTORY: Certain function in the ART translation needed to be
;**             automated. This is an attempt at that automation.
;**    INSTALLATION: TI Explorer.
;**                  iPSC Hypercube.
;**
;**
;**
;*****************************************************************************
;*****************************************************************************


;;; function to add a clause to a list (rule)

(defun attach (plist field)
  (cond
    ((null plist)
     (setq plist (list field)))
    (t (setq plist (append plist (list field))))))
```

```
;;;  function that only works on TI Explorer to find variables that begin with "?"

(defun replaceq (field objlist)
  (cond ((numberp field) field)
        ((atom field)
         (cond
           ((string-equal field '? :start1 0 :start2 0 :end1 1 :end2 1)
            (cond ((equal field (member field objlist)) '(< ,field))
                  (t (cons field objlist) '(> ,field))))
           ((or (string-equal field 'S?) (string-equal field 'S) '+)
           ((not (null field)) field)))
        ((null (cdr field)) (list (replaceq (car field) objlist)))
        (t (cons (replaceq (car field) objlist)
                 (replaceq (cdr field) objlist)))))

;;; function that works with replaceq to find variables with "?".

(defun parse (field objlist)
  (setq field (replaceq field objlist))
  (cond ((equal (car field) 'not)
         (setq field (list (first (cadr field))
                           (second (cadr field))
                           (list (first (third (cadr field)))
                                 (list '- (second (third (cadr field))))))))
        (t field)))

;;; function to change an expression from in-fix to pre-fix notation

(defun inf-to-pre (e)
  (let (a-list)
    (cond ((atom e) e)
          ((setq a-list (match '((> v)) e nil))
           (inf-to-pre (match-value 'v a-list)))
          ((setq a-list (match '((+ l) (restrict ? oneplus) (+ r)) e nil))
           '(+ ,(inf-to-pre (match-value 'l a-list))
               ,(inf-to-pre (match-value 'r a-list))))
          ((setq a-list (match '((+ l) - (+ r)) e nil))
           '(- ,(inf-to-pre (match-value 'l a-list))
               ,(inf-to-pre (match-value 'r a-list))))
          ((setq a-list (match '((+ l) * (+ r)) e nil))
           '(* ,(inf-to-pre (match-value 'l a-list))
               ,(inf-to-pre (match-value 'r a-list))))
          ((setq a-list (match '((+ l) / (+ r)) e nil))
           '(/ ,(inf-to-pre (match-value 'l a-list))
               ,(inf-to-pre (match-value 'r a-list))))
          ((setq a-list (match '((+ l) ^ (+ r)) e nil))
           '(expt ,(inf-to-pre (match-value 'l a-list))
                  ,(inf-to-pre (match-value 'r a-list))))
          ((setq a-list (match '(- (+ r)) e nil))
           '(- ,(inf-to-pre (match-value 'r a-list))))
          (t e))))
```

```
;;; function needed for previous routine to identify addition

(defun oneplus (x)
  (equal x '+))

;;; find the value of a key into an association list.

(defun match-value (key a-list)
  (cadr (assoc key a-list)))

;;; function to transform an ART rule into a rule usable by the inference engine

(defun change-rule (rule)
  (cond ((not (equal (car rule) 'defrule)) (print "not a rule"))
        (t (setq rule-name (cadr rule))
           (setq new-rule nil)
           (let ((new-rule nil)
                 (if-part nil)
                 (then-part nil))
             (do ((fields (cddr rule) (cdr fields))
                  (then nil)
                  (objlist nil)
                  (declare nil))
                 ((null fields) new-rule)
               (cond
                 ((equal (car fields) '=>) (setq then t))  ;;sense andecedant
                 ;((atom (car fields)) nil) ;; gets rid of comments
                 ((atom (car fields))       ;; keeps comments
                  (cond
                    (then (setq then-part
                                (append (list (car fields)) then-part)))
                      (t    (setq if-part
                                (append (list (car fields)) if-part)))))
                 ((equal (caar fields) 'declare)          ;; get salience
                  (setq declare t)
                  (setq new-rule (list 'rule (list rule-name
                                                   (list (first (cadar fields))
                                                  (inf-to-pre
                                                  (second (cadar fields)))))))))
                 ((and (not (equal (caar fields) 'declare))
                                                    ;; add salience if not there
                       (equal declare nil))
                  (setq declare t)
                  (setq new-rule (list 'rule (list rule-name '(salience 0)))))
                 ((equal (caar fields) 'schema)
                  (setq if-part (attach if-part (parse (car fields) objlist))))
                 ((equal (caar fields) 'modify)
                  (setq then-part
                       (attach then-part (parse (car fields) objlist))))
                 ((equal (caar fields) 'not)
                  (setq if-part (attach if-part (parse (car fields) objlist))))
                 ((equal (caar fields) 'assert)
```

```
                    (setq then-part
                        (attach then-part (parse (car fields) objlist))))
                   (t (cond (then (setq then-part
                                        (append (list (car fields)) then-part)))
                            (t    (setq if-part
                                        (append (list (car fields)) if-part))))
                      (print (car fields)))
                   ))
          (setq new-rule (append new-rule
                                 (list (append '(if) if-part))
                                 (list (list 'then then-part))))
          (add-rule new-rule)


;;; function to find the slot name

(defun slot (field)
  (car (caddar field)))

;;; function to find the schema name

(defun schema (field)
  (cadar field))

;;; function to find the value of a slot of a schema

(defun value (field)
  (cadr (caddar field)))

;;; function to add a rule to the list of rules

(defun add-rule (newrule)
  (cond ((null rules)
         (setq rules (list newrule))
         newrule)
        (t (setq rules (cons newrule rules))
           newrule)))

;;; function to change ART facts to a form compatible with the inference engine

(defun change-facts (fact)
  (cond ((not (equal (car fact) 'deffacts)) (print "not a fact"))
        ( t (do ((facts (cddr fact) (cdr facts))
                 ((null facts) t)
             (add-fact (car facts))))))

;;; function to change ART globals to a form compatible with the inference engine

(defun change-global (global)
  (cond ((not (equal (car global) 'defglobal)) (print "not a global"))
        (t (eval `(setq ,(cadr global) ,(cadddr global))))))

;;; function to write the transformed rules out to a file for a file transfer between
```

```
;;; systems


(defun write-rules (filename)
  (setf out-stream (open filename :direction :output))
  (do ((rule rules (cdr rule)))
      ((null rule) t)
    (pprint (car rule) out-stream))
  (close out-stream))

;;; initializes a list of schema names

(setq schemas nil)

;;; function to transform ART schemas into a form compatible with the inference
;;; engine.

(defun change-schema (schema)
  (cond ((not (equal (car schema) 'defschema))
         (print '(name ,(cadr schema) is not a schema)))
        (t
         (setq schemas (cons (cadr schema) schemas))
         (add-defaults-schema schema)
         (let ((schemaname (cadr schema))
               (schemabody (cddr schema))
               (clauses    nil))
           (do ((body schemabody (cdr body)))
               ((null body) clauses)
             (cond
               ((listp (car body)) (setf (get schemaname (first (car body)))
                                         (second (car body))))))))))

;;; function to add default slots to the schemas for the RAV

(defun add-defaults-schema (schema)
  (let  ((instance-of (cadr (assoc 'instance-of (cdddr schema))))
         (schemaname (cadr schema)))
    (cond ((equal instance-of 'plan)
           (setf (get schemaname 'bindings) '(empty))
           (setf (get schemaname 'globals) 'error-in-initializing-this-plan)
           (setf (get schemaname 'step) 0)
           (setf (get schemaname 'time) 0)
           (setf (get schemaname 'abort-plan) nil)
           (setf (get schemaname 'abort-plan-stack) '(empty))
           (setf (get schemaname 'number-of-steps) 1000)
           (setf (get schemaname 'timer) nil))
          ((equal instance-of 'need)
           (setf (get schemaname 'importance) *default-salience*)
           (setf (get schemaname 'bindings) '(empty))
           (setf (get schemaname 'bindings-updated) nil)
           (setf (get schemaname 'status) 'inactive)
           (setf (get schemaname 'preselected-plan) nil)
           (setf (get schemaname 'timer) nil)
```

```
        (setf (get schemaname 'failed-plan)
              '(slot-how-many multiple-values))
        (setf (get schemaname 'succeeded-plan)
              '(slot-how-many multiple-values))
        (setf (get schemaname 'parameters) nil))
       ((equal instance-of 'numeric-valued-object)
        (setf (get schemaname 'possible-value) 'numeric)
        (setf (get schemaname 'time-of-last-update) 0)
        (setf (get schemaname 'exact-value) 0)
        (setf (get schemaname 'quantization) 1)
        (setf (get schemaname 'tolerance) 0)
        (setf (get schemaname 'value) 0)
        (setf (get schemaname 'old-value) 0)
        (setf (get schemaname 'vcs-target-value) -1000059)
        (setf (get schemaname 'target-value) nil)
        (setf (get schemaname 'type) 'unset)
        (setf (get schemaname 'broken) 'no))
       (t (print `(instance-of ,instance-of))))))
```

Hypercube Specific Code

```
;;*****************************************************************************
;;*****************************************************************************
;;**
;;**
;;**    TITLE: Hypercube Unique Inference Engine.
;;**    DATE: 7 December 1987.
;;**    VERSION: 1.0.
;;**    FUNCTION: To provide the concurrent match for the parallel inference
;;**              engine.
;;**    LANGUAGE: Concurrent Common LISP.
;;**    INPUTS: None.
;;**    OUTPUTS: None.
;;**    FILES READ: None.
;;**    FILES WRITTEN: None.
;;**    AUTHOR: Donald J. Shakley.
;;**    HISTORY: Uniquely developed for this thesis.
;;**    INSTALLATION: AFIT iPSC Hypercube.
;;**
;;**
;;**
;;*****************************************************************************
;;*****************************************************************************



;;; variables used for timing and counting the rules fired

(setq match-time 0)
(setq select-time 0)
(setq act-time 0)
(setq match-start-time 0)
```

```
(setq match-stop-time 0)
(setq select-start-time 0)
(setq select-stop-time 0)
(setq act-start-time 0)
(setq act-stop-time 0)
(setq rule-count 0)

;;; function to initialize the previous variables upon each execution of the inference
;;; engine.

(defun initial-setting ()
   (setq match-time 0)
   (setq select-time 0)
   (setq act-time 0)
   (setq match-start-time 0)
   (setq match-stop-time 0)
   (setq select-start-time 0)
   (setq select-stop-time 0)
   (setq act-start-time 0)
   (setq act-stop-time 0)
   (setq rule-count 0))

;;; function that is the main body of the parallel inference engine

(defun forward-chain ()
 (initial-setting)
 (do ((done T))
     ((null done) (not done))
    (setq agenda nil)
    (setq match-start-time (sys:clock))
    (do ((rules-to-try rules (cdr rules-to-try))
           (rules-tried 0))
           ((null rules-to-try) t)
           (cond ((use-rule (car rules-to-try))
                  ;;(print '(rule ,(car rules-to-try) **** USED ******))
                  (setq rules-tried (+ rules-tried 1)))
           ;((> rules-tried 25) (setq rules-tried 0)
                        ; (print '(more rules tried ,(length rules-to-try) left)))
          (t (setq rules-tried (+ rules-tried 1))) ))
     (setq match-stop-time (sys:clock))
     (setq match-time
          (+ match-time (- match-stop-time match-start-time)))
     (setq select-start-time (sys:clock))
     (select-a-rule agenda)
     (setq select-stop-time (sys:clock))
     (setq select-time (+ select-time (- select-stop-time select-start-time)))
     (setq act-start-time (sys:clock))
     (cond ((null selected-rule) (setq done nil))
           ((null (car selected-rule)) (setq done nil))
           (t (setq rule-count (+ rule-count 1))
            (remember selected-rule)))
     (setq act-stop-time (sys:clock))
     (setq act-time (+ act-time (- act-stop-time act-start-time))))
```

```
    (print '(Number of rules fired  ,rule-count))
    (print '(match time  ,match-time  milliseconds))
    (print '(select time  ,select-time milliseconds))
    (print '(act time  ,act-time milliseconds)))

;;; function that selects the overall rule to fire by receiving the rules from its children
;;; (if it has any), using these rules along with the rules on its own agenda to select
;;; a rule to pass to its parent. The node then waits to receive the overall rule selected
;;; from its parent

(defun select-a-rule (agenda-list)
  (append (get-children) agenda-list)
  (setq selected-rule (car agenda-list))
  (print '(agenda length ,(length agenda-list)))
  (do ((rules-to-select agenda-list (cdr rules-to-select)))
      ((null rules-to-select) selected-rule)
    ;(print '(rule ,(caar (car rules-to-select)) has salience
          ;,(salience (car rules-to-select))))
    (cond ((< (salience selected-rule) (salience (car rules-to-select)))
           (setq selected-rule (car rules-to-select)))))
  ;(print '(before sending to parent selected rule is ,selected-rule))
  (send-to-parent selected-rule)
  (cond ((= (sys:mynode) 0) selected-rule)
        (t (setq selected-rule (recv-overall-rule)))))

;;; function to receive the selected rule from each of its children

(defun get-children ()
  (do ((children (find-children) (cdr children))
       (rule-list nil))
      ((null children) rule-list)
      (cond ((< (car children) (expt 2 (sys:cubedim)))
             ;(print '(waiting to receive child ,(car children)))
             (setq rule-list (cons (wait-recv (car children))
                                   rule-list)))
             ;(print '(received child: ,rule-list)))
            (t (setq children nil)))))

;;; function to send selected overall rule to a node's children

(defun send-to-children (rule)
  (do ((children (find-children) (cdr children)))
      ((null children) rule)
      (cond ((< (car children) (expt 2 (sys:cubedim)))
             (let ((out-strm
                (make-fasl-node-stream (car children)
                   :tree :direction :output :element-type 'string-char)))
             (unwind-protect
             (progn
                (cond
                   ((null rule) (send out-strm :dump-object '(nil)))
                   (t (send out-strm :dump-object rule)))
                (funcall out-strm :finish-output))
```

```
                (send out-strm :close))))
              (t (setq children nil)))))

;;; function to send a node's parent its selected rule

(defun send-to-parent (rule)
 (cond ((= (sys:mynode) 0) (send-to-children rule))
       (t
         (let ((out-strm
            (make-fasl-node-stream (find-parent)
               :tree
               :direction :output
               :element-type 'string-char)))
           (unwind-protect
           (progn
             ;(print '(sending to parent: ,(find-parent)))
             ;(print '(sending rule: ,rule))
             (cond
                  ((null rule) (send out-strm :dump-object '(nil)))
                  (t (send out-strm :dump-object rule)))
             (funcall out-strm :finish-output))
             (send out-strm :close))))))

;;; function to receive and pass on the overall rule

(defun recv-overall-rule ()
         (send-to-children (wait-recv (find-parent))))

;;; function to find a node's children

(defun find-children ()
  (cadr (assoc 'children (cadr (assoc (sys:mynode) spanning-tree)))))

;;; function to find a node's parent

(defun find-parent ()
  (cadr (assoc 'parent   (cadr (assoc (sys:mynode) spanning-tree)))))

;;; synchronous message receive

(defun wait-recv (node)
 (let ((in-strm
       (make-fasl-node-stream node
        :tree
        :direction :input
        :element-type 'string-char)))
    (unwind-protect
    (progn
      (do ((to-exit nil)
           (ret-strm nil))
          (to-exit ret-strm)
          (cond
            ((null (setq ret-strm (send in-strm :read-object)))
```

```
            (setq to-exit nil))
           (t (setq to-exit t)
              ;(print '(wait-recv ,ret-strm))
              (send in-strm :close))))))))

;;; variable to define the spanning tree

(defvar spanning-tree)

;;; table used to define the spanning tree's connections

(setq spanning-tree
 '(
    ( 0  ((children (1 2 4 8 16)) (parent nil)))
    ( 1  ((children (3 5 9 17))   (parent 0)))
    ( 2  ((children (6 10 18))    (parent 0)))
    ( 3  ((children (7 11 19))    (parent 1)))
    ( 4  ((children (12 20))      (parent 0)))
    ( 5  ((children (13 21))      (parent 1)))
    ( 6  ((children (14 22))      (parent 2)))
    ( 7  ((children (15 23))      (parent 3)))
    ( 8  ((children (24))         (parent 0)))
    ( 9  ((children (25))         (parent 1)))
    ( 10 ((children (26))         (parent 2)))
    ( 11 ((children (27))         (parent 3)))
    ( 12 ((children (28))         (parent 4)))
    ( 13 ((children (29))         (parent 5)))
    ( 14 ((children (30))         (parent 6)))
    ( 15 ((children (31))         (parent 7)))
    ( 16 ((children nil)          (parent 0)))
    ( 17 ((children nil)          (parent 1)))
    ( 18 ((children nil)          (parent 2)))
    ( 19 ((children nil)          (parent 3)))
    ( 20 ((children nil)          (parent 4)))
    ( 21 ((children nil)          (parent 5)))
    ( 22 ((children nil)          (parent 6)))
    ( 23 ((children nil)          (parent 7)))
    ( 24 ((children nil)          (parent 8)))
    ( 25 ((children nil)          (parent 9)))
    ( 26 ((children nil)          (parent 10)))
    ( 27 ((children nil)          (parent 11)))
    ( 28 ((children nil)          (parent 12)))
    ( 29 ((children nil)          (parent 13)))
    ( 30 ((children nil)          (parent 14)))
    ( 31 ((children nil)          (parent 15)))
))
```

Bibliography

Aho, Alfred V., John E Hpcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley Publishing Company, 1974.

Albus, James Sacra. *Brains, Behavior, and Robotics*. Peterborough, NH: BYTE Publications Inc, 1981.

ART 3.0. Reference Manual. Inference Corporation, Los Angeles, CA, January 1987.

Baer, Jean-Loup. *Computer Systems Architecture*. Rockville, MD: Computer Science Press, Inc., 1980.

Barr, Avron and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence, Volume I*. Standford, CA: HeurisTech Press, 1981.

Billstrom, David, Joseph Brandenburg, and John Teeter. "CCLISP on the iPSC Concurrent Computer," *Proceedings of the National Conference on Artificial Intelligence* 7-12 (1987).

Blair, Jesse and Karl E. Schricker. "Robotic Air Vehicle: A Pilot's Perspective." Unpublished report. Air Force Wright Aeronautical Laboratories, Wright-Patternson AFB, OH, 1986.

Blair, Jesse. Project Manager, Robotic Air Vehicle. Personal Interview. Air Force Wright Aeronautical Laboratories, Wright-Patterson, OH, May 1987.

Booch, Grady. *Software Components with Ada*. Menlo Park, CA: Benjamin/Cummings Publishing Company, Inc, 1987.

Brandenburg, Joseph E. and David S. Scott. *Embeddings of Communication Trees and Grids into Hypercubes*. iPSC Technical Report No 1. Intel Scientific Computers, Beaverton, OR, 1986.

Broekhuysen, Martin, editor. *Concurrent Common Lisp User's Guide Version 1.1*. Cambridge, MA: Gold Hill Computers, 1987a.

-----. *Concurrent Common Lisp Reference Manual Version 1.1*. Cambridge, MA: Gold Hill Computers, 1987b.

Cheng, P. Daniel and J. Y. Juang. "A Parallel Resolution Procedure Based on Connection Graph," *Proceedings of the National Conference on Artificial Intelligence* 13-17 (1987).

Douglass, Robert J. "A Qualitative Assessment of Parallelism in Expert Systems." *IEEE Software, 2*: 70-81 (May 1985).

Evans, David J. editor. *Parallel Processing Systems*. Cambridge, Great Britain: Cambridge University Press, 1982.

Explorer. Lisp Reference (2243201-0001). Texas Instruments Incorporated, Austin, TX, June 1985.

Fanning, Jesse. System Software Technician. Personal Interview. Air Force Wright Aero nautical LAboratories, Wright-Patterson AFB, OH, November, 1987.

Fischler, Martin A. and Oscar Firschein. *Intelligence: The Eye, the Brain, and the Computer.* Reading, MA: Addison-Wesley Publishing Company, 1987.

Forgy, Charles and others. "Initial Assessment of Architecures for Productior Systems," *Proceedings of the National Conference on Artificial Intelligence* 116-120 (1984).

-----. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Prob lem," *Aritficial Intelligence 19:* 17-37 (1982).

Gevarter, William B. *Artificial Intelligence, Expert Systems, Computer Vision, and Natural Language Processing.* Park Ridge, NJ: Noyes Publications, 1984.

Graham, Joyce M., Engineer, Artificial Intelligence Laboratory. Personal interview. Texas Instruments Defense and Electronics Group, Dallas, TX, May 1987.

Gupta, Anoop. *Parallelism in Production Systems.* PhD dissertation. Carnegie-Mellon Uni versity, Pittsburgh, PA, March 1986.

Hillis, Daniel W. "The Connection Machine." *Scientific American 256*: 108-115 (June 1987).

Hwang, Kai. "Advanced Parallel Processing with Supercomputer Architectures." *Proceed ings of the IEEE 75:* No 10 1348-1379 (October 1987).

Hwang, Kai and Faye A. Briggs. *Computer Architecture and Parallel Processing.* New York: McGraw-Hill Book Company, 1984.

Ishida, Toru and Salvatore J. Stolfo. "Towards the Parallel Execution of Rules in Produc tion System Programs," *Proceedings of the International Conference on Parallel Processing* 568-575 (1985).

Jamieson, Leah H., Dennis B. Gannon, and Robert J. Douglass editors. *The Characteristics of Parallel Algorithms.* Cambridge, MA: MIT Press, 1987.

Kelly, Michael A. and Rudolph E. Seviora. "A Multiprocessor Architecture for Production Systems," *Proceedings of the National Conference on Artificial Intelligence* 36-41 (1987).

Kornfeld, William A. "The Use of Parallelism to Implement a Heuristic Search," *Proceed ings of the 7th International Joint Conference on Artificial Intelligence* 575-580 (1981).

Li, Guo-jie and Benjamin W. Wah. "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms," *Proceedings of International Conference on Par allel Processing* 473-480 (1984).

-----. "How Good Are Parallel and Ordered Depth-First Searches?" *Proceedings of the International Conference on Parallel Processing* 992-999 (1986).

Lystad, Garr S. "The TI Dallas Inference Engine (TIDIE) Knowledge Representation Sys

tem," *Proceeding of the IEEE National Aerospace and Electronics Conference* 1348-1351 (May 1987).

Manna, Zohar. *Mathematical Theory of Computation*. New York, NY: McGraw-Hill Book Company, 1974.

McNulty, Christa. "Knowledge Engineering for Piloting Expert System," *Proceedings of the IEEE National Aerospace and Electronics Conference* 1326-1330 (May 1987).

Minsky, Marvin L. *Computation: Finite and Infinite Machines*. Englewood Cliffs, NJ: Pren tice-Hall Inc, 1967.

Miranker, Daniel P. "TREAT: A Better Match Algorithm for AI Production Systems," *Proceedings of the National Conference on Artificial Intelligence* 42-47 (1987).

Moler, Cleve and David S. Scott. *Communication Utilities for the iPSC*. iPSC Technical Report No 2. Intel Scientific Computers, Beaverton, OR, 1986.

Mraz, Capt Richard T. *Performance Evaluation of Parallel Branch and Bound Search with the Intel iPSC Hypercube Computer*, MS Thesis AFIT/GCE/ENG/86D-2. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1986.

Nilsson, N. J. *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga Publishing Company, 1980.

Norman, Capt Douglas O. *Reasoning in Real-Time for the Pilot Associate: An Examination of Model Based Approach to Reasoning in Real-Time for Artificial Intelligence Systems using a Distributed Architecture*, MS Thesis AFIT/GCS/ENG/85D-12. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1985.

Oflazer, Kemal. "Partitioning in Parallel Processing of Production Systems," *Proceedings of the International Conference on Parallel Processing* 92-100 (1984).

Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley Publishing Company, 1984.

-----. "Heuristic Search Theory: Survey of Recent Results," *Proceedings of the 7th Inter national Conference on Artificial Intelligence* 554-562 (1981).

Post, Emil L. "Formal Reductions of the General Combinatorial Decision Problem," *American Journal of Mathematics, 65* 197-268 (1943).

Rich, Elaine. *Artificial Intelligence*. New York, NY: McGraw-Hill Book Company, 1983.

Saridis, George N. "Intelligent Robotic Control," *IEEE Transactions on Automatic Control, AC-28* 547-557 (May 1983).

Stolfo, Salvatore J. "Five Parallel Algorithms for Production System Execution on the DADO Machine," *Proceedings of the National Conference on Artificial Intelligence* 300-307 (1984).

Wah, Benjamin W., Guo-jie Li and Chee Fen Yu. "Multiprocessing of Combinatorial Search Problems," *Computer* 93-108 (June 1985).

Ward, Paul T. and Stephen J. Mellor. *Structured Development for Real-Time Systems Volume 1: Introduction and Tools.* New York, NY: Yourdon Press, 1985.

Waterman, Donald A. *A Guide to Expert Systems.* Reading, MA: Addison-Wesley Publish ing Company, 1986.

Winston, Patrick Henry. *Artificial Intelligence.* Reading, MA: Addison-Wesley Publishing Company, 1984.

Winston, Patrick Henry and Berthold Klaus Paul Horn. *LISP.* Reading, MA: Addison-Wesley Publishing Company, 1984.

VITA

Captain Donald J. Shakley was born 24 September 1960 in Petrolia, Pennsylvania. He graduated from high school in Karns City, Pennsylvania, in 1978 and attended Purdue University, from which he received the degree of Bachelor of Science in Computer Science and Mathematics in August 1982. Upon graduation, he received a commission in the USAF through the ROTC program. He was employed as a computer consultant for Witco Chemical Corporation, Petrolia, Pennsylvania, until called to active duty in October 1982. He served as a computer software programmer/analyst with the 2045th Communications Group, Andrews AFB, MD, until entering the School of Engineering, Air Force Institute of Technology, in May 1986.

Permanent address: Box 73

Petrolia, PA 16050

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/87D-24	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFWAL 8b. OFFICE SYMBOL (If applicable) AFWAL/AAI	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Air Force Wright Aeronautical Laboratories Wright-Patterson AFB OH 45433	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO
11. TITLE (Include Security Classification) See Box 19	
12. PERSONAL AUTHOR(S) Donald J. Shakley, B.S., Capt, USAF	
13a. TYPE OF REPORT MS Thesis 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1987 December 15. PAGE COUNT 110
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	09	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial Intelligence, Parallel Processing, Real-Time

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: Parallel Artificial Intelligence Search Techniques for Real-Time Applications

Thesis Chairman: Gary B. Lamont, PhD

Approved for public release: IAW AFR 190-1.

LYNN E. [illegible]
[illegible] Development
[illegible]
Wright-[illegible]

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Gary B. Lamont, PhD	22b TELEPHONE (Include Area Code) (513) 255-3576 22c OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Abstract

State space search is an important component of many problem solving methodologies. The computational models within Artificial Intelligence depend heavily upon state space searches. Production systems are one such computational model. Production systems are being explored for real-time environments where timing is of a critical nature. Parallel processing of these systems and in particular concurrent state space searching seems to provide a promising method to increase the performance of production systems in the real-time environment.

Production systems in the form of expert systems, for example, are being used to govern the intelligent control of the Robotic Air Vehicle (RAV) which is currently a research project at the Air Force Wright Aeronautical Laboratories. Due to the nature of the RAV system, the associated expert system needs to perform in a demanding real-time environment. The use of a parallel processing capability to support the associated computational requirement may be critical in this application. Thus, parallel search algorithms for real-time expert systems are designed, analyzed and synthesized on the Texas Instruments (TI) Explorer and Intel Hypercube. Examined is the process involved with transporting the RAV expert systems from the TI Explorer, where they are implemented in the Automated Reasoning Tool, to the iPSC Hypercube, where the system is synthesized using Concurrent Common LISP (CCLISP). The performance characteristics of the parallel implementation of these expert systems on the iPSC Hypercube are compared to the TI Explorer implementation.

The implementation on the iPSC hypercube points out the feasibility of implementing a production system in CCLISP and gaining performance improvements over the TI Explorer. This study shows poor performance speedups due to poor load balancing combined with a large communication overhead in contrast to the problem size.

END
DATE
FILMED
4-88
DTIC